

We have now reached that point where we are going to assume that you know how to load a program into the system, execute it, single-step it, and load and examine the various registers in the CPU. From this point on, programs will be documented only by the flow diagrams and program listings that have been accompanying the text. You have already seen and used these program listings in the last five chapters, but before going on, we'd like to pause and discuss some of the formalities of those listings.

The first column of the listing always contains the address of the memory location holding the instruction. This is expressed in hexadecimal, but because most computer systems do not follow the address with an H when printing out a listing of this sort, we too will omit the H. The second column is the hex code of the instruction or data occupying that location. Again the H is omitted. These two columns contain all the information you need to load your program into the computer via the DCM mode.

Even though columns one and two are sufficient to document your programs, adding more information to the listing will make it much easier to follow. This is done in columns three and four. Column three is used for the mnemonic of the instruction represented by the hex code in column two. After all, with all of these hex codes it would be very difficult to just look at a long column of hex numbers and read or troubleshoot a program. So column three tells us what the hex codes stand for. Notice that instructions requiring two or three memory locations have nothing in this column after the entry opposite the first location. This is so that the complete instruction mnemonic, including data or address, is on one line so that it is easier to read. Thus MVI D, CFH appears on one line even though the data itself, CFH, is at the next memory location. See Figs. 6-1 and 6-2.

We can make a program much easier to follow if we leave ourselves some notes. Convention in the programming world requires that these notes, called comments, are separated from the rest of the program by a semicolon. This eliminates any confusion between mnemonics and comments.

Remember, columns three and four are for reference purposes only. To load a program we need to know only the addresses and the hex codes of the instructions to go into those addresses, and that information is contained in columns one and two. But by including the mnemonics of the instructions in column three, we can "read" the program ourselves, and if we have explanatory comments in column 4 to denote what the various program segments are used for, we should be able to come back to the program listing six months later and still be able to understand it.

COLUMN 1 ADDRESS	2 CONTENTS	3 MNEMONIC	4 COMMENT
0100	3E	MVIA, FBH	;STORE DATA IN
0101	FB	STA 6000H	;ACCUM.
0102	32		;STORE CONTENTS OF
0103	00		;ACCUM AT 6000H.
0104	60		;
0105	CD	CALL 300H	;CALL DALAY.
0106	00		;
0107	03		;

Fig. 6-1. Format of the program listing. Column 2 contains the data or instruction that is contained in the corresponding address in column 1. Column 3 is an English-language equivalent of

column 2. Column 4 allows us to write comments about the program, so that we can understand its purpose and flow quickly the next time we work with it.

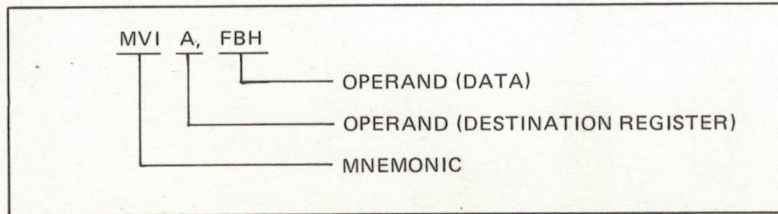


Fig. 6-2. Actually, as we see in this figure, the term mnemonic only applies to that part of the instruction that tells the computer what action to perform. The data, address-

ses and register names supplied in the instruction are called operands. Thus, in the instruction LDA 6000H, the mnemonic is LDA and the operand is 6000H.

To assist you in the writing of programs, we have included a machine language programming pad. On it you will find preprinted blank columns so that you can write and save your programs conveniently. This pad is three-hole punched on the right side of the paper instead of the conventional left side. That way, after you are an expert programmer, you can take this text out of the binder and replace it with your programming pad. By keeping the pad on the left side of the binder, you can write programs on the pad and execute and debug them at the same time using the keyboard and displays on the right.

Along these same lines, we will, from time to time, ask you to try out your programming skills. In these cases, shorter versions of the machine language programming sheets are included at appropriate place within the text so that you can write and execute your own programs.

Four Groups of Instructions. We have discussed and used quite a few of the instructions in the ia7301 instruction set. Before we get too confused, let's stop and discuss some of the groups these instructions fall into. That way we'll begin to think of instructions as types rather than hundreds of individual, unrelated instructions. Along these lines, we have already divided the instruction set into four broad categories. In the first five chapters of this text we have used various instructions to illustrate the text, and as it turns out, we have already become familiar with at least one instruction in each of these categories.

Data Transfer Group. The first category is the Data Transfer Group. We have used quite a few instructions that fit into this category. So far, they are:

Mnemonic	Hex Code
MVI A,D8	3E
MVI B,D8	06
MVI C,D8	0E
MVI D,D8	16
LXI B,D16	01
LDA A16	3A
STA A16	32
STAX B	02
STAX D	12

In this little table, we have used D8 to denote two hex digits of data, D16 to denote four hex digits of data, and A16 to denote a four hex digit address.

As the name implies, Data Transfer instructions are used to transfer data from one part of the computer to another. This applies whether we are discussing registers in the CPU or locations in the system memory. Since much of the operations of the computer consist of moving data around from register to memory, register to register, memory to register, register to I/O ports, etc., data transfer instructions make up a large subset of the total computer instruction set.

We can think of the computer system as being made up entirely of memory locations, and strictly speaking, this is true. Oh, some of the locations can do more than just store data. The accumulator, for instance, can perform certain arithmetic operations on the data that we store there, but all of these locations store data as their most basic function. The instruction set operates by moving data from one location to another and then modifying the data stored at the various locations.

While this simplistic view of the computer is true, it tends to limit our ability to group the instructions into useful categories. For this purpose it is much better to break the concept of memory locations into two types. The first is the conventional memory location that makes up the entire system memory. These locations are unable to perform any arithmetic or logical operations on the data they store. In addition, the large number of these locations makes it necessary to specify a four hex digit address in order to store or retrieve data. The second type is the working register contained in the CPU. While these working registers can be used to store data in the same way as the regular memory locations, they have the added ability of being able to operate on the data they store. The number of these working registers is limited, in fact, this is one of the ways we can evaluate the usefulness of a new CPU. The more working registers a CPU has, the easier it will be to move and operate on data within the computer. The relatively small number of registers has one benefit, since now many operations can be specified using a highly abbreviated address. This makes data transfers from register to register much simpler and faster than data transfers from one memory location to another.

Keeping the distinction between memory locations and registers in mind, let's review some of the instructions we have discussed. Any data transfer requires that the source and the destination of the data be specified by the instruction. The Direct Addressing method incorporates this information right into the instruction itself. We have already used the STA A16 instruction, which causes the contents of the accumulator to be stored at the memory location specified by the address A16. The source of the data, the accumulator, is contained within the mnemonic of the instruction, STA. The destination of the data is specified by the address A16. The instruction requires three program memory locations to hold it, one for the mnemonic and two for the address. It is an example of a register to memory type of data transfer. The LDA A16 instruction is another variation of this type of transfer. Both are directly addressed since they specify both source and destination of the data within the confines of the instruction itself. See Fig. 6-3.

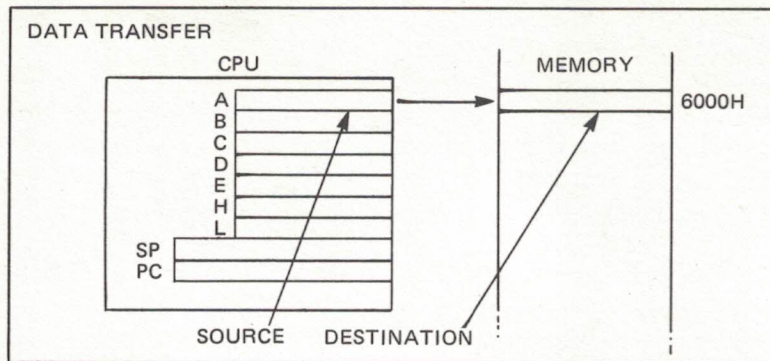


Fig. 6-3. Every data transfer involves a source and a destination; the instruction must specify both. In the case of STA 6000H, the instruction itself states the source as the accumulator (STA is the mnemonic for Store Accumulator). The destination, in this case memory location 6000H, is stated as an address in conjunction with the mnemonic. The entire instruction requires six hex

digits, 32H — the hex code for STA, and 6000H — The address of the memory location that will be the destination of the data. To store the complete instruction in the memory as part of a program to be executed will require these memory locations. This is an example of a Direct Address instruction since both source and destination are specified by the instruction.

The card at the front of your binder contains all of the instructions that can be executed by the ia7301. Examine it as you read this chapter. The card is divided into two sections. The top section is a listing, organized by ascending hex code, of the entire instruction set. This listing is normally used for identifying an instruction when only the hex code is known. This situation occurs from time to time in debugging a program that is already contained in the memory.

It is the second, lower section that is the most interest to us now. Here are the various instructions, grouped as to function, and their corresponding hex codes. One of the largest of these groups is the Data Transfer Group. There, in a small block labeled Direct, you will find LDA A16 and STA A16. After each of the instruction is the appropriate hex code, 3A in the case of

LDA, and 32 for STA. By referring to this card as we write and execute our programs, we can quickly locate the hex code for any instruction we wish.

It is possible to cause a data transfer directly from one register to another. This uses a class of direct addressing instructions called MOV X,Y. These cause data transfers to take place between Y, the source register, and X, the destination register. This is another example of a direct addressed data transfer since both source and destination are specified by the instruction.

The largest block within the Data Transfer Group is the MOV instruction block. Since each of these instructions must specify both source and destination registers, the MOV block is set up in the form of a matrix. To use it find the destination register along the top of the matrix. Then follow the correct column down until you reach the line corresponding to the desired source register. Within the square where column and row overlap you'll find the hex code. Thus the hex code for MOV A,B is 78H. When expressing the source and destination, the destination is always listed first. MOV A,B is properly read as Move the contents of B to A. This listing order tends to be confusing to many people, but it is too well established to invent your own system. Some people have found it easier to read the instruction by substituting the word "load" for "move." MOV A,B thus becomes Load register A with the contents of register B. See Fig. 6-4.

We have seen that LDA A16 and STA A16 both require three program memory locations in order to specify a data transfer that is a memory to register type. The MOV A,B type of instructions can effect a register to register data transfer with only one memory location required for the instruction. Here is the real worth of the working registers. There are few enough of them that we can effect a data transfer between any two registers with a short instruction that can be stored in a single programming location. Since these instructions are short and execute quickly, we will use them as much as possible in our programs. In fact, as you gain in programming skills one of the first plateaus that you will reach is the ability to maneuver your data around within the registers

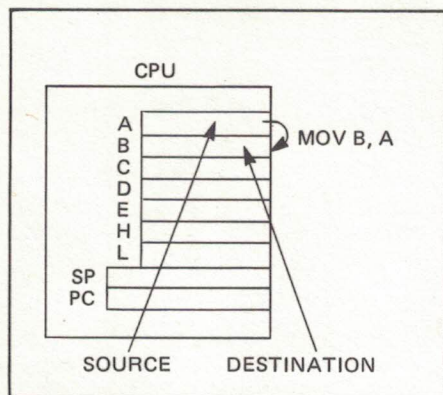


Fig. 6-4. Register to register types of data transfers are easier to specify, but still require a source and a destination. The direct addressed form of this type of data transfer is the MOV instruction. It specifies both source and destination in the form MOV B, A and requires only one memory location to store the hex code (47H). There is a different hex code for every possible combination of source and destination registers. Use of register to register instructions rather than register to memory types wherever possible will result in shorter, faster programs.

to minimize the number of register to memory data transfers which require lots of program memory and so much time to execute.

A second way to cause a data transfer is the Immediate method. Immediate instructions are those that contain the data for the transfer within the instruction. MVI A,FFH causes the accumulator to be loaded with the data FFH. Since the data is contained in the instruction, there is no source register as in the other types of transfers that we have been discussing; the instruction itself is the source of the data. The destination register must be specified by the instruction. In this sense MVI A can be thought of as both an Immediate and a Directly Addressed instruction. Although we have only used four of the MVI instructions, there are a total of eight, all of which are shown in the Immediate block within the Data Transfer Group. These types of transfers require two memory locations to specify the instruction, one for the instruction itself and one for the data. The Immediate instructions do not require the amount of memory that the register to memory directly addressed instructions did, but they do require more than the register to register group, MOV. See Fig. 6-5.

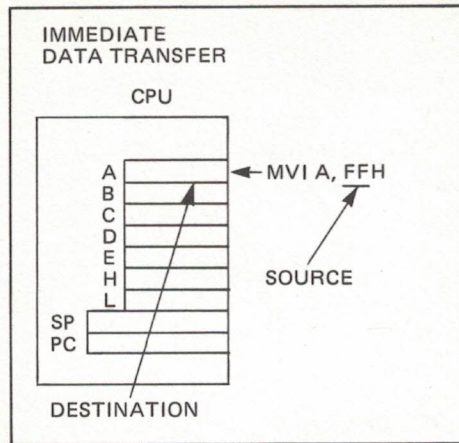


Fig. 6-5. The immediate Data Transfer replaces the source register with the instruction itself. Thus the instruction supplies data which is loaded into the destination register.

The third form of data transfer is the Indirectly Addressed method. STAX B is an example of an indirect instruction; you'll find it along with some others in the Indirect Block of the Data Transfer Group on the hex card. In this form of instruction, either the source or destination register is specified by the instruction directly. In the case of STAX B, the source register is directly specified to be the accumulator. The destination is specified, not by the instruction, but by the contents of the accumulator to be stored at that memory location addressed by the contents of the B and C register pair. While this form of addressing may, at first, seem awkward, it is potentially one of the most powerful methods. By setting up a loop for sending the contents of the accumulator to memory via the STAX instruction, and then incrementing the register pair so that on each pass through the loop, a new, adjacent memory location is addressed by the register pair, we can move large blocks of data around within the system without having to use a large amount of program memory.

There are three additional blocks of instructions within the Data Transfer Group on the hex card. While these do not represent other types of data transfers, they do have some ingredient that makes it convenient to group them together. One of these blocks is made up of only instructions involving register pairs. We have already discussed LXI RP, which is really an Immediate instruction, but it is easier to group it with the rest of the register pair data transfers. Another block is made up of a special set of instructions that involve saving the contents of the registers in the memory in much the same way that a CALL instruction uses the memory to save the return address. All of these instructions involve the stack pointer, and hence, are grouped under Stack Operations. They will be discussed shortly. The last group is made up of special Input/Output instructions that would be used if the ia7301 were not a memory mapped I/O system. Since it is, these instructions are useless in this system and are reprinted here for reference purposes only.

With the memory mapped I/O structure, as in the ia7301, many of the data transfer instructions that we have been discussing can be used to move data into and out of the I/O ports.

The Arithmetic/Logic Group. Once we have moved the data to the desired place within the computer by means of a data transfer instruction, it is time to operate on the data. In its simplest form this can be a simple increment or decrement operation. In more complex instructions, the operation may be an Exclusive-Or or any of many other operations. The key to this group is that the contents of the register or memory location are modified by the instruction. For instance the contents of the register are different after performing the incrementing operation. The instructions that we have used from this group are listed below:

Mnemonic	Hex Code
INR A	3CH
DCR B	05H
DCR C	0DH
DCR D	15H
DCR E	1DH

In the Direct block of arithmetic/logic instructions, we find the eight basic operations that are performed on the data in the accumulator. Since each of these operations must also involve a data word, it is necessary to specify where that data will be found. Thus the Direct block contains a matrix of operations and source registers, so that we pick the operation we wish to perform by scanning down the vertical list of instructions. Once found, we scan across the horizontal line to find the source register that will supply the data for the operation. ADD C, for instance, will ADD the contents of the C register to the contents of the accumulator. The result will replace the previous contents of the accumulator; the C register will be unchanged by the operation. The hex code for this operation is 81H.

Notice that this is not really a data transfer; the contents of the C register play a part in the operation, but only in the sense that they partially determine how the contents of the accumulator are changed. It is possible to do all of the eight arithmetic and logical operations with a set of Immediate instructions, the second block within the Arithmetic/Logic Group. The end effect is the same, the only difference being the source of the data used to modify the contents of the accumulator. It is possible to perform operations on the contents of the accumulator without resorting to other registers or outside data. These are the Accumulator instructions. This block consists only of those instructions that operate on the contents of the accumulator without introducing new data. These involve some complementing operations, rotating and shifting operations, some that involve only the Carry Flag, considered here to be part of the accumulator, and a very special instruction called DAA which is used to convert binary (and hexadecimal) numbers into decimal numbers.

Of course, there is no reason why arithmetic and logical operations can only be done on the contents of the accumulator. Granted, that register is set up to perform the widest variety of these kinds of operations, but to a lesser degree, the other registers can be used too. Each of these registers or register pairs can be incremented or decremented and there is even a special addition operation between register pairs that can be executed. All of these blocks together make up the Arithmetic/Logic group.

Change of Control Instructions. Normally the computer follows the program as it is stored in memory, executing the first instruction, then the second instruction, and so forth. Change the order in which the instructions are stored and the order of execution will change. The program counter register of the CPU keeps track of which instruction is to be executed by storing the address of that instruction. Normally the program counter acts as a simple counter. Each time the computer executes an instruction, the program counter increments itself by one, two or three, depending upon the number of locations required to store the instruction.

Much of the power of the computer lies in its ability to repeat simple operations over and over again without having to store the instructions over and over again. This is done by looping and the use of subroutine calls. Both of these techniques have been explored in previous chapters. In each case they operate by loading a new address into the program counter so that program execution continues at a different point in the program. In the case of a JMP instruction, the prior contents of the program counter are lost. However when a CALL instruction is executed, the contents of the program counter are saved in one of the memory locations. Then when a RET instruction is executed, the earlier contents of the program counter will be replaced so that program execution picks up where it left off. Each of these instructions is an example of the Transfer of Control Group. So far we have used five of these in the earlier chapters.

Mnemonic	Hex Code
JMP A16	C3H
JNZ A16	C2H
CALL A16	CDH
RET	C9H
RZ	C8H


Control Instructions. There are a few instructions that are used to control the computer in the sense that they control the way the computer will respond to certain types of activities. They frequently result in special control signals being sent to certain I/O ports, etc. The only one of these instructions that we have used so far is the RST 7. In fact, this instruction is frequently classified as a Transfer of Control instruction since it causes a jump to another portion of the program. It results however, from a special set of signals being jammed onto the data bus whenever you press the **STEP** key. The way we have used RST 7 as a program stopper is not the normal usage of this instruction. Normally, the instruction never actually occurs in the program memory. Instead, the **STEP** key jams the hex code for this instruction onto the data bus overriding what was originally there. The CPU thinks it sees the RST 7 coming back from memory and it executes it. Thus, we can cause a jump to another portion of the program (the step portion of the system monitor) any time we wish. Other instructions in this group cause the computer to halt, to enable or disable certain types of I/O activities, and to perform no operation (no operation, it turns out, can be a very handy thing to do!).

In the upcoming chapters, we will cover all of the instructions in the four basic groups, one by one. The purpose of this discussion was two-fold. First to acquaint you with the organization of the hex card; you're going to be seeing a lot of it from now on. Second, to tie together some of the concepts embodied in the instructions we have already learned.

At this point you should understand the way the program is stored and executed out of memory. Each individual instruction should fall into one of four groups. It either moves data around within the system, operates on that data, goes to a different place within the program or performs some sort of control function on the computer. We can move data around in three different ways. We either specify the source and destination of the data in the instruction, we eliminate the need for specifying the source by including the data in the instruction itself, or we identify another set of registers that hold the address of the data.

Practice will clear up any confusion that may be remaining as to the nature of the various groups and the addressing methods they use. So let's get back to real programming.

Performing Conversions. Let's see. When we last left our program for driving the display digits in Chapter Five, we had successfully lighted the left-most display, albeit not with the number that we had anticipated. It seems that seven-segment displays will not accept hexadecimal numbers and display them properly.

Each hexadecimal number must be converted to a new number that the displays will accept. For instance, if we send a hexadecimal 5 to the displays, they will indicate a figure as yet unknown to the western world... . A moment's reflection should explain the problem. A hexadecimal 0 should result in that figure appearing in the display. But a zero obviously requires that certain segments should be lighted, and the number of segments, which happens to be four, does not correspond in any way to the numeric value of the number that we sent there, zero. Obviously, the hex numbers have to be converted to some other form so that a zero, for instance, will produce a zero on the displays.

It is our contention that sending a COH to the displays will cause a zero to appear. That is, the correct conversion number for a hex 0 is COH, and we can prove it. Is your program from Chapter Five still in memory? Check it using the **DCM** key against the program listing below.

0100	16	MVI D, CFH	;Load D register with first half of
0101	CF		;address CFXXH. E can be ignored at
0102	3E	MVI A, 00H	;this time since only two digits
0103	00		;of address are used. Load accum-
0104	12	STAX D	;ulator with data 00H and write to
0105	CD	CALL DELAY 1	;data register, Delay.
0106	00		;
0107	03		;
0108	CD	CALL DELAY 2	;Delay to correct duty cycle for
0109	00		;other seven displays not being used.
010A	02		;
010B	C3	JMP 0104H	;Jump back to 0104H to close loop.
010C	04		;This way display will remain lighted.
010D	01		;
0300	06	MVI B, 4FH	;Load register B with timing constant.
0301	4F		;This value has been chosen to simulate
0302	05	DCR B	;the guard circuit timing out.
0303	C2	JNZ 0302H	;If B is not zero, go back and decrement
0304	02		;again. If zero, return to calling
0305	03		;program.
0306	C9	RET	;
0200	0E	MVI C, 07H	;Set up register C as loop counter.
0201	07		;

0202	CD	CALL DELAY 1	;Display 1 simulates each of the
0203	00		;other digits. The loop counter
0204	03		;causes this to happen seven times.
0205	0D	DCR C	;Decrement loop counter.
0206	C8	RZ	;If zero, return to the calling
0207	C3	JMP 0202	;program. If not, go back to 020H.
0208	02		;
0209	02		;

This program operates by writing the data in the accumulator, loaded by the MVI instruction at 0102H and 0103H, to the data register. DELAY 1 simulates the time for the LED guard circuit to time out. DELAY 2 simulates the time for the other seven displays to display their data. Since the guard circuit operates for the other seven displays as well, the program for simulating these displays also makes use of DELAY 1. Thus DELAY 1 is a subroutine CALL within the subroutine DELAY 2, This is a subroutine within a subroutine; this practice is called subroutine NESTING, and is common in any program of any length.

When the second subroutine is CALLED, the contents of the program counter are stored in the memory and the stack pointer decremented by two as before. This means that there are two return addresses in the stack, and that locations 00FCH, 00FDH, 00FEH and 00FFH are all tied up storing return addresses. This portion of the memory is called the stack. It grows downward from location 0100H just as the program grows upward from that same location. There is no reason why the concept of nesting cannot be extended to many levels, and most significant programs will do this.

Since it is our intention to test whether or not loading C0H into the display register will produce a zero on the display, we need to replace the 00H in the MVI A, 00H at locations 0102H and 0103H with C0H. Do this with the DCM key so that location 0103H contains C0H. Now reset

the program counter with **CLR** and execute the program again. This time you should see a zero in the left most display.

There is a hex code for each of the hex numerals that will cause the seven-segment displays to display the correct numeral. We could write down a table of these codes for you, but you already have that table in your possession. You see, the ia7301 uses that table to look up the conversion codes in the same way we would. The table is stored at location 8282H to 8291H, and if you look in those locations you'll find it. In the first location, 8282H is the code for zero, which as we have seen, is COH. In the next is the code for one, the next contains the code for two, the next contains the code for three, and so on, one location for each of the hex digits 0-F. By using the **DCM** key to examine these locations, you can produce your own table of hex code/seven-segment equivalents. Fill in the blanks below.

Address	Hex Numeral	Seven-Segment Equivalent
8282H	0	COH
8283H	1	_____
8284H	2	_____
8285H	3	_____
8286H	4	_____
8287H	5	_____
8288H	6	_____
8289H	7	_____
828AH	8	_____
828BH	9	_____
828CH	A	_____
828DH	B	_____
828EH	C	_____
828FH	D	_____
8290H	E	_____
8291H	F	_____

Once this table is filled in, we can use it to try other digits. If, for instance, you substitute the value for hex A into location 0103H, and executed the program again, you'll see the letter A displayed on the LEDs. Try it.

Automatic Table Look-Up. We can automate the lookup process by a technique called Indexed Addressing. Since we're going to be using this subroutine a lot, we'll put it up higher in the memory where it'll be out of the way of our regular program. We'll assume that when this table look-up subroutine is called, the accumulator will contain a hexadecimal digit that we wish to convert to the seven-segment equivalent. The flow diagram of the subroutine is shown in Fig. 6-6.

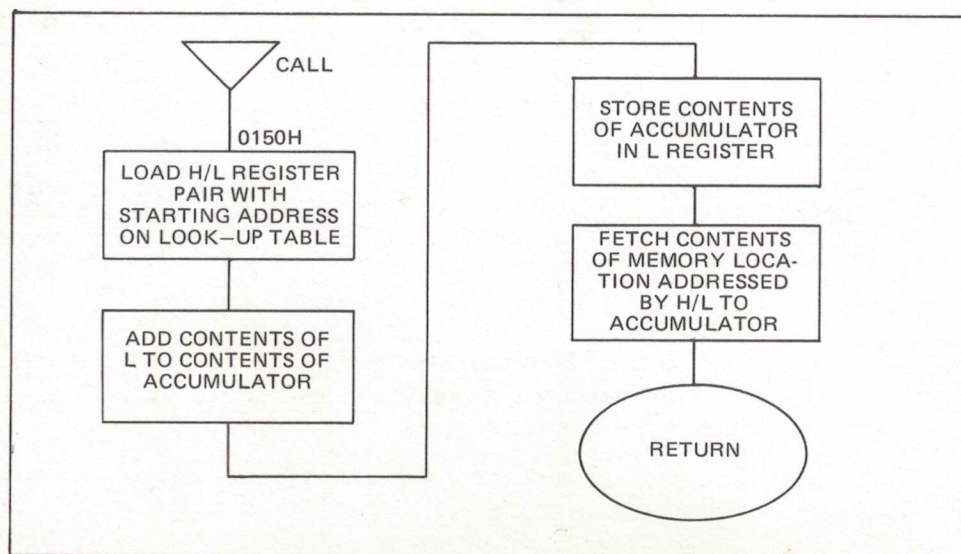


Fig. 6-6 Automatic table look-up.

The first step is to load the H and L registers with the address of the first location in the segment table, 8282H. The fastest way to do this is with an LXI H instruction, hex code 21H. That instruction operates in the same fashion as the LXI B, D16 instruction that we discussed in the last chapter. The four hex digits represented by D16 are loaded into the specified register pair, in this case H and L. The two most significant digits go into H and the two least significant digits into L. You will find this instruction on your hex card in the Data Transfer Group within the Register Pair Block.

Once the H and L register pair has been loaded with the starting address of the table, we will need to modify it so that the register pair contains the address, not of the first location, but of the location containing the codes corresponding to the hex digit in the accumulator. This is easily done by adding the contents of the L register to the contents of the accumulator. The accumulator then contains the least two significant digits of the address of the seven-segment equivalent of the former contents of the accumulator. The contents of the accumulator are then moved into the L register replacing the former contents of that register. H and L now contain a new address, that of the table location corresponding to the hex number formerly in the accumulator. This address can be used to read that table and move the contents of that location to the accumulator. We have thus managed to replace a hex number in the accumulator with its seven-segment equivalent.

We have covered the operation of the table look-up subroutine rather quickly and there are some holes that need explaining. The quickest way to do this is by loading the program into your computer and executing it step by step, talking our way through it as we go. Load the following program:

0150	21	LXI H, 8282H	;Point the H and L registers to the
0151	82		;starting address in the conversion table.

0152	82		;
0153	85	ADD L	;Add the contents of the L register
0154	6F	MOV L,A	;to the accumulator. Move the contents
0155	7E	MOV A,M	;of the accumulator to L. Use H and L
0156	FF	RST 7	;to fetch the contents of the table
			;to the accumulator.

Let's try the program with some known data. First, load the accumulator with 00H using the DCR mode. Then continue to step through the registers with the **NXT** key until the program counter is displayed. Load this with 0150 so that when **STEP** or **EXC** is pressed, the program will execute the first instruction at location 0150H. If this is not done, the program will execute from location 0100H because of the automatic program counter reset feature of the **CLR** key.

Before we try the first instruction, let's take a look at the contents of the H and L registers. After all, that first instruction is an LXI H which is going to cause those registers to be loaded with the new data. If we don't know what the prior contents of the registers are, we'll never know if the contents were changed by the instruction. Use the **DCR** key to check the contents of those registers and write the results below:

Contents of L register _____
 Contents of H register _____

Now press **STEP** and reexamine the contents of the registers.

New Contents of L register _____
 New Contents of H register _____

If the instruction executed properly, the L register should now contain 82H and the H register should also contain 82H. This is because the starting address of the table just happens to be symmetrical, that is, the two most significant digits just happen to match the two least significant digits. This is a coincidence; normally the L and H registers would be loaded with different data. In any event, we can see that the LXI H instruction operates in the same fashion as LXI B discussed earlier. LXI H thus joins our growing repertoire of instructions.

Press **STEP** and execute the next instruction, the ADD L at location 0153H. This operates by adding the contents of the L register to the contents of the accumulator and storing the results back in the accumulator. Since the accumulator originally contained 00H and the L register contains 82H, ADD L should result in the total of 00H + 82H, or 82H being stored in the accumulator. This is easy to check by pressing **DCR** to examine the accumulator. It now contains 82H. This now represents the least two significant digits of the address. To be used the contents of the accumulator must be moved to the L register. Then the contents of the H and L registers can be treated as an address to fetch the contents of the proper table entry.

Press **STEP** to execute the next instruction, MOV L, A at location 0154H. We have already discussed this class of instructions but this is the first time we have used one in a program. It operates by duplicating the contents of the accumulator into the L register. We use the word "duplicate" rather than the instruction mnemonic, Move, because Move conjures up a vision of data moving from register to register. If it is moved from the accumulator to the L register, it must not be in the accumulator anymore, and of course, that is not what happens. In fact, data is duplicated into the destination register, but remains unchanged in the source register. As we have already pointed out, the registers are listed in the mnemonic with the destination register preceding the source register. This tends to confuse new programmers since the tendency is to read the instruction MOV L, A as Move L to A which is incorrect. Some programmers get around this by reading MOV as Load, so that MOV L, A is read Load L with the contents of A.

The MOV instructions represent an entire block of instructions within the Data Transfer Group on the hex card. They make up a matrix organized by source and destination registers so that the hex code for any combination of registers can quickly be found. We are going to pass on to the next instruction without demonstrating MOV L, A at this point, because for the initial choice of 00H for the contents of the accumulator, both registers reach this point in the program containing 82H and the MOV L, A accomplishes nothing. We will return to it in a moment.

The next instruction, at location 0155H, is another MOV. This differs in that it is an example of indirect addressing whereas the other MOV instructions are all direct addressing. Direct addressing implements the data transfer by specifying source and destination of the data right in the instruction itself. For instance, LDA A16 specifies the destination, the accumulaotr, in the mnemonic and the source by the address A16. MOV L, A is an example of direct addressing since both source and destination are specified in the instruction.

The MOV instruction at 0155H, MOV A,M, only looks like another direct addressed instruction. The destination is specified to be the accumulator by the letter A. The source is specified to be the memory by the letter M. Ah, but what location in memory? For that, we have to look at the H and L registers, for they contain the address of the memory location that will be referenced by the instruction. This then, is an indirect addressed instruction since the source for the data transfer is specified, not by the instruction, but by the H and L registers.

Even though MOV A,M is an indirect instruction it is included in the MOV matrix along with all of the direct addressed instructions, simply because that's where everyone expects to find it. We can see this instruction execute. The accumulator now contains 82H. Press **STEP** and then check it again with the **DCR** key. It will now contain COH, which, as we saw earlier when we prepared a listing of the look-up table, is the data at 8282H corresponding to a hex digit input of 00H. The routine has thus successfully been entered with the accumulator containing a hex 0 and exited with the correct seven-segment code, COH.

Let's try it again, Load a 05H into the accumulator and reset the program counter to 0150H with the **DCR** and **NXT** keys. Press **STEP** to load the H and L registers with 8282H, the starting address of the table. The L register now contains 82H and the accumulator 05H. Press **STEP** again. We can see the effect of the MOV L,A instruction at 0154H by using **DCR** to check the L register. It should now contain 87H having had the data in the accumulator duplicated into the L register. This completes the address to be referenced by the next instruction, MOV A,M. Press **STEP**. Using the **DCR** key verify the contents of the accumulator. It should contain 92H, which as we saw when we made the listing of the look-up table, and then indexes to the correct point within the table.

Free-Running Counter. Now that we have a way to perform the table look-up directly, let's put it into a useful program. The problem is to make the computer count by combining the routines we have already worked out for driving the left-most display and the routine for automatic table look-up. The flow diagram for the program is illustrated in Fig. 6-7

Since we want to start the counter at 0, the first instruction is a MVI A, 00H to initialize the process. We are going to be using the accumulator to perform the table look-up, a process that destroys the previous contents of the accumulator. Since we want the counting process to be continuous, it will be necessary to store the contents of the accumulator so that they can later be retrieved and reloaded. This is the purpose of the LDA 0050H instruction at location 0102H. We have thus defined location 0050H as the temporary storage location for the contents of the accumulator. The table look-up routine at location 0150H is then called to perform the conversion from hex digit in the accumulator to seven-segment code. Notice that the hex digit is lost in the process; it will later be recovered from memory location 0050H. Use of the table look-up as a subroutine requires that we replace the RST 7 program stopper at location 0156H with a RET instruction so that the main program can be reentered when the subroutine is completed.

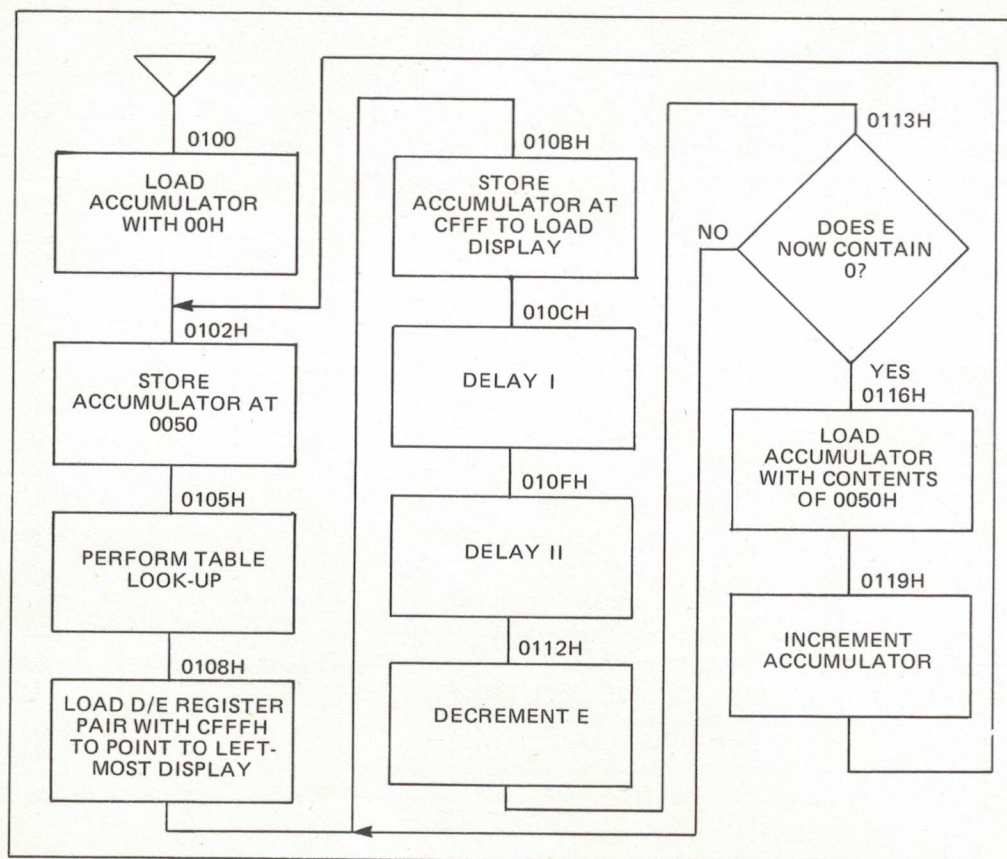


Fig. 6-7 Flow diagram of program to make the computer count.

When the look-up subroutine is completed, the accumulator will contain the seven-segment code ready to drive the display. The D and E registers are then loaded with the address of the left-most display, CF, and a timing constant, FFH. Used in this manner, the D register will be used to address the display and the E register will be used as the timer to define how long each digit will be displayed before it is incremented. By using an LXI D instruction, both registers can be loaded at once. A STAX D instruction causes the contents of the accumulator to be sent to the display. CALLing DELAY I simulates the time for the guard circuit to time out and CALLing DELAY II simulates the other seven displays. The E register is then decremented and tested by the JNZ instruction at 0113H to see if the digit has been displayed for the correct time. If the E register still contains a non-zero number, program execution returns to the STAX D instruction where the display is hit once more with a brief pulse of energy coded with the correct data to produce the display 0. If, on the other hand, the E register contains zero, it is time to perform the count. The accumulator no longer has the current value having lost that number during the table look-up. The correct number exists, however, in location 0050H; we can retrieve it with a LDA 0050H instruction. An INR A instruction causes the value to be incremented which produces the counting action. All that remains is to perform a jump back to location 0102H which stores the new incremented value of the accumulator at location 0050H and the action starts over.

Load the program below into the computer. Don't forget the return instruction at 0156H. You can executed it by pressing **CLR** and **EXC**.

0100	3E	MVIA,00H	;Load the accumulator with zero to
0101	00		;start the counter at zero.
0102	32	STA 0050H	;Store the count at location 0050H.
0103	50		;
0104	00		;

0105	CD	CALL LOOK UP	;Perform the table look-up.
0106	50		;
0107	01		;
0108	11	LXI D, CFFFH	;Load D with address of display and
0109	FF		;E register with the timing constant
010A	CF		;FFH.
010B	12	STAX D	;Send the contents of the accumulator
010C	CD	CALL DELAY I	;to the display. DELAY I to simulate
010D	00		;the guard circuit.
010E	03		;
010F	CD	CALL DELAY II	;Call DELAY II to simulate the time
0110	00		;for the other seven displays.
0111	02		;
0112	1D	DCR E	;Decrement the timing counter.
0113	C2	JNZ 010B	;Has timer timed out? If not, go back
0114	0B		;and send data to display again. If
0115	01		;zero, go on and load the accumulator
0116	3A	LDA 0050H	;with the current count from 0050H.
0117	50		;
0118	00		;
0119	3C	INR A	;Count.
011A	C3	JMP 0102H	;Complete the loop.
011B	02		;
011C	01		;
0150	21	LXI H, 8282H	;Point H and L to the starting address
0151	82		;of the table.
0152	82		;
0152	85	ADD L	;Add the least two significant digits
0154	6F	MOV L,A	;of the address to the accumulator. Move
0155	7E	MOV A,M	;the contents of the accumulator to L.
0156	C9	RET	;Use H and L to fetch the contents
			;of the table to the accumulator.

0200	0E	MVI C,07H	;Set up register C as loop counter.
0201	07		;
0202	CD	CALL DELAY I	;Delay I simulates each of the
0203	00		;other digits. The loop counter
0204	03		;causes this to happen seven times.
0205	0D	DCR C	;Decrement loop counter.
0206	C8	RZ	;If zero, return to the calling
0207	C3	JMP 0202	;program. If not, go back to 0202H
0208	02		;
0209	02		;
0300	06	MVI B,4FH	;Load register B with timing constant.
0301	4F		;This value has been chosen to simulate
0302	05	DCR B	;the guard circuit timing out.
0303	C2	JNZ 0302H	;If B is not zero, go back and decrement
0304	02		;again. If zero, return to calling
0305	03		;program.
0306	C9	RET	;

You should see the left-most display light immediately with the digit 0, followed quickly by 1, 2, 3, and so forth. If you continue to watch the counting action, the count will progress beyond 9 to A, B, C, and through the hexadecimal digits. Once F is reached, the counting action will continue with random symbols. This is because the indexed addressing portion of the program does not recognize that the range of numerals 0-F has been exceeded. It continues to count and try to convert this count to seven-segment equivalents by indexing up the conversion table. We could correct this fairly easily, but let's go on to more interesting problems. Later in this chapter, we'll correct the counting process and even add some new wrinkles when we program the computer to be a digital clock.

Operating All of the Displays. Now that we have had a small measure of success driving one of the displays, let's stretch our luck and try driving all of them at once. If we wanted to display the same character in all eight of the displays, this would be a relatively simple matter. Since this would not be a particularly useful thing to do, we're going to complicate matters by trying to write the program so that each of the displays is independent and can indicate a different character. That way, the displays can be used for real numbers, up to eight digits in length. To simplify matters temporarily, we are going to ignore the fact that the data needs to be converted to seven-segment code to appear on the displays, but we'll correct that shortly.

Because this subroutine is going to be very useful in some of the later programs, we'll want to place it high in the memory so that it will not interfere with the main program that begins at location 0100H and works up. The subroutine, called DISPLAY, begins at location 0250H. It requires that we maintain two sets of addresses at once. First, the display drivers must be addressed. You will recall that the addresses of these displays are C1XXH, C3XXH, C5XXH, C7XXH, C9XXH, CBXXH, CDXXH and CFXXH for the digits, reading from right to left. We will have to address each of these displays in turn if we are to light them all. Then too we'll have to fetch the data to be displayed from somewhere in memory. In our last program, this was simply loaded into the accumulator via a direct instruction. But for this program, we'll want to use memory locations to store the data to be displayed. That way, calculations can be performed and the results stored in these locations and DISPLAY called to display them. We have arbitrarily chosen locations 0200H through 0207H to hold this data, with location 0207H corresponding to the left-most display, and so on. These eight locations must be addressed at the same time the corresponding display digit is being addressed. The program to be presented here is one method of handling both addresses at once; we'll see others later in this chapter.

The first step is to point the D/E register pair to the left-most display digit. The address for this digit is CFXXH, where the X's represent hex digits that may take on any value without affecting the execution of the program. We could load the D/E register pair with an LXI D instruction, but this requires three memory locations. Since the contents of the E register are immaterial, being X's, we can ignore E altogether and point the register pair to the display digit by loading only the D register. This is done with a MVI D, CFH instruction which only requires two memory locations. Once this has been done we can send data to the display digit. First, however, the correct data must be fetched from the memory. This can be done with a LDA 0207H which loads the accumulator with the contents of memory location 0207H, which represent the data to appear in the left-most display. The contents of the accumulator are then sent to the display with a STAX D instruction. This is followed by CALLING DELAY I to simulate the timing out of the guard circuit which completes the first digit display, and we can move on to the second. Since the address of the second digit is CDXXH, which is two less than the address of the left-most digit, CFXXH, we need to decrement the D/E register pair by two so that they point to the next digit. Notice that DELAY II is no longer needed to simulate the time to pulse the over seven display digits. Since this problem is going to actually perform that action, simulation is not needed.

We could continue and develop the entire program, for pulsing the other seven digits but at some point you're going to have to try writing your own program and this is a good place to begin. Use the blanks below to complete the program following the guidelines we established for the first digit. Since the displays need to be pulsed repeatedly to be visible, we will end the program with a jump back to 0250H to set up a loop. After you have filled in the program the way you believe it should be, flip the page and compare your program with ours. No new instructions are used.

0250	16	MVI D,CFH	;Point D/E pair to CFXXH.
0251	CF		;
0252	3A	LDA 0207H	;Fetch data from memory.
0253	07		;
0254	02		;
0255	12	STAX D	;Send data to display.
0256	CD	CALL DELAY I	;Simulate guard circuit timing out.
0257	00		;
0258	03		;
0259	15	DCR D	;Decrement D by two to point D/E to
0251	15	DCR D	;next digit.
025B	_____	_____	;
025C	_____	_____	;
025D	_____	_____	;
025E	_____	_____	;
025F	_____	_____	;
0260	_____	_____	;
0261	_____	_____	;
0262	_____	_____	;
0263	_____	_____	;
0264	_____	_____	;
0265	_____	_____	;
0266	_____	_____	;
0267	_____	_____	;
0268	_____	_____	;
0269	_____	_____	;
026A	_____	_____	;

026B	_____	_____	:
026C	_____	_____	:
026D	_____	_____	:
026E	_____	_____	:
026F	_____	_____	:
0270	_____	_____	:
0271	_____	_____	:
0272	_____	_____	:
0273	_____	_____	:
0274	_____	_____	:
0275	_____	_____	:
0276	_____	_____	:
0277	_____	_____	:
0278	_____	_____	:
0279	_____	_____	:
027A	_____	_____	:
027B	_____	_____	:
027C	_____	_____	:
027D	_____	_____	:
027E	_____	_____	:
027F	_____	_____	:
0280	_____	_____	:
0281	_____	_____	:
0282	_____	_____	:
0283	_____	_____	:
0284	_____	_____	:
0285	_____	_____	:

```

0286 _____ ;
0287 _____ ;
0288 _____ ;
0289 _____ ;
028A _____ ;
028B _____ ;
028C _____ ;
028D _____ ;
028E _____ ;
028F _____ ;
0290 _____ ;
0291 _____ ;
0292 _____ ;
0293 _____ ;
0294 _____ ;
0295 _____ ;
0296 _____ ;
0297 _____ ;
0298 C3 JMP 0250 ;Complete the loop.
0299 50 ;
029A 01 ;

```

To operate correctly, your program must load the accumulator with the data in the memory registers used for storing the display data. This data is then sent to the displays, a delay implemented to simulate the guard circuit timing out, and the next display addressed. Since the address of each display digit differs from the next by a count of two, to go from one display to the next it will be necessary to decrement the D/E register pair by two. If we were to send the data in the

accumulator to each of the display digits, without changing the contents of the accumulator as each display is accessed, the same number would appear in all of the displays. Since we wish to use the eight display digits for different numerals so that eight digit numbers can be presented, it is not enough to simply send the same data to each of the display digits. Thus the accumulator must be loaded with the data appropriate to the digit being accessed before its contents are sent to the digit. This means that the LDA instruction must access a different memory location for each of the digits, 0207H for the first, 0206H for the second and so on.

When you think that your program is correct, you can try executing it by using the **DCR** and **NXT** keys to display the program counter, and the loading 0250H with the hex keys. Don't forget to press **NXT** once more after you have loaded the PC display so that the data in the displays actually goes into the real program counter. Once this is done, press **EXC** to execute your program.

Did it work? If so, there's a possibility it did not work. Worse yet, when a program does not work, it has a very annoying habit of going off and destroying itself and any other programs that may also be in the memory. One reason this happens is a error in keying in the jump or call addresses. This usually occurs after the preliminary portions of the program have been written, and the need is found to add or subtract an instruction. This changes the length of the subroutine or program segment, which will in turn change the starting addresses of other program subroutines or segments. When these starting addresses are changed, any other instructions that reference them must also be changed. A good way to keep track of these instructions is to star them with a red pencil in the margin of your program sheet, on star next to each instruction that references another memory location. This is espically important for any kind of jump or subroutine call instructions. Then when instructions are added or deleted, it is a simple matter to scan down the programming sheet checking to be sure that the addresses that appear within the starred instructions are modified to reflect the changes in the program.

If this is not done, program execution will follow the incorrect jump or call to the wrong address. This is bad enough when that address is the first location of an instruction, but imagine what happens when the jump to location is the second or third location of a multiple location instruction like LDA. The computer tries to execute the data at that location as though it were the first location of an instruction. Depending on exactly what that data is, all sorts of things can happen. Usually the system executes the data as though it were a program and continues until it finds data that causes it to loop back to some earlier point. In this case the displays simply go blank. Sometimes a single character will appear on the displays, march across them four times and then disappear completely. Occasionally the computer will find an FFH somewhere in its travels, and when that happens, a jump to the single-step part of the computer monitor occurs and the displays come on with the STEP display. Of course, this causes the computer to stop executing the pseudoprogram that it found.

Earlier in this book we followed a practice of placing an FFH at the end of each program segment that did not have a jump or a return instruction at the end of it. This was to prevent the computer from going on past the end of our little program segment and chasing around at random in the memory. You see, when that happens, there is a very good chance that sooner or later it will find some instructions that causes data to be written into the memory right where our program was. Then we would have to go through the program, location by location looking for the incorrect entry, for once this over-writing has occurred, the program will not execute properly.

To summarize these two points before going back to our program: ALWAYS STAR PROGRAM INSTRUCTIONS THAT REFERENCE OTHER MEMORY LOCATIONS, AND ALWAYS PLACE AN FFH PROGRAM STOPPER AT THE END OF EACH PROGRAM SEGMENT THAT DOES NOT HAVE A JUMP OR RETURN TYPE OF TERMINATOR.

We wanted to digress for a moment to explain what types of difficulties might have happened if you chose to execute your trial program. In any event, compare your program against ours. How did you fare?

0250	16	MVI D, CFH	;Point D/E pair to CFXXH.
0251	CF		;
0252	3A	LDA 0207H	;Fetch data from memory.
0253	07		;
0254	02		;
0255	12	STAX D	;Send data to display
0256	CD	CALL DELAY I	;Simulate guard circuit timing out.
0257	00		;
0258	03		;
0259	15	DCR D	;Decrement D by two to point D/E to
025A	15	DCR D	;next digit.
025B	3A	LDA 0206H	;Fetch data from memory for next digit.
025C	06		;
025D	02		;
025E	12	STAX D	;Send data to next display digit.
025F	CD	CALL DELAY I	;Simulate guard circuit timing out.
0260	00		;
0261	03		;
0262	15	DCR D	;Decrement D by two to point D/E to
0263	15	DCR D	;next digit.
0264	3A	LDA 0205H	;Fetch data from memory for next digit.
0265	05		;
0266	02		;
0267	12	STAX D	;Send data to next display digit.
0268	CD	CALL DELAY I	;Simulate guard circuit timing out.
0269	00		;
026A	03		;
026B	15	DCR D	;Decrement D by two to point D/E to

026C	15	DCR D		;next digit.
026D	3A	LDA	0204H	;Fetch data from memory for next digit.
026E	04			;
026F	02			;
0270	12	STAX D		;Send data to next display digit.
0271	CD	CALL DELAY I		;Simulate guard circuit timing out.
0272	00			;
0273	03			;
0274	15	DCR D		;Decrement D by two to point D/E to
0275	15	DCR D		;next digit.
0276	3A	LDA	0203H	;Fetch data from memory for next digit.
0277	03			;
0278	02			;
0279	12	STAX D		;Send data to next display digit.
027A	CD	CALL DELAY I		;Simulate guard circuit timing out.
027B	00			;
027C	03			;
027D	15	DCR D		;Decrement D by two to point D/E to
027E	15	DCR D		;next digit.
027F	3A	LDA	0202H	;Fetch data from memory for next digit.
0280	02			;
0281	02			;
0282	12	STAX D		;Send data to next display digit.
0283	CD	CALL DELAY I		;Simulate guard circuit timing out.
0284	00			;
0285	03			;
0286	15	DCR D		;Decrement D by two to point D/E to
0287	15	DCR D		;next digit.
0288	3A	LDA	0201H	;Fetch data from memory for next digit.
0289	01			;
028A	02			;
028B	12	STAX D		;Send data to next display digit.

028C	CD	CALL DELAY I	;Simulate guard circuit timing out.
028D	00		;
028E	03		;
028F	15	DCR D	;Decrement D by two to point D/E to
0290	15	DCR D	;next digit.
0291	3A	LDA 0200H	;Fetch data from memory for next digit.
0292	00		;
0293	02		;
0294	12	STAX D	;Send data to next display digit.
0295	CD	CALL DELAY I	;Simulate guard circuit timing out.
0296	00		;
0297	03		;
0298	C3	JMP 0250H	;Complete the loop.
0299	50		;
029A	02		;

Check to be sure this program and the DELAY I subroutine at location 0300H are loaded properly in the memory. Then set the program counter to 0250H with the **DCR** key and execute the program. You should see all of the displays light up, although the characters that appear will probably not be recognizable. This is because we did not convert them to seven-segment code. To prove this, use the DCM mode to enter the following data into memory:

0200	C0
0201	F9
0202	A4
0203	B0
0204	99
0205	92
0206	82
0207	F8

You probably recognize these characters from the table that we prepared earlier for the seven-segment code conversion table. They represent the seven-segment codes for the numerals 0-7. By manually loading them into locations 0200H through 0207H, we are forcing the program to display digital numbers instead of garbage.

CLR	n-----
DCR NXT NXT NXT NXT	uPC-----
NXT NXT NXT NXT NXT	uPC-0250
0 2 5 0	uA-----
NXT	76543210
EXC	

We have successfully written and executed a program that drives all eight of the seven-segment displays. Before we go on to bigger and better things, replace the C3H instruction at location 0298H with a C9H (Return). This now sets up the display program segment as a callable subroutine so that we can use it as part of other programs.

Let's take a moment and review the programs now contained in the computer memory. First, there is a subroutine residing at location 0250H that converts the contents of the accumulator into its seven-segment code equivalent. Return occurs with the code in the accumulator replacing the data that was there originally. Then we have a subroutine at location 0200H which causes the contents of memory locations 0200H-0207H to appear in the displays. Since we have replaced the JMP at the end of this program with a RET, this subroutine no longer loops continually but is only passed once. Since this would not make the displays visible, the subroutine

must be called continuously to make the displays light. Finally, at location ⁰³⁰⁰0200H is a short delay subroutine used to simulate the guard circuit of the displays timing out.

We will now try to strengthen the conversion subroutine at location 0150H so that it operates with the data contained in memory locations 0200H-0207H. In doing this we will add another subroutine on top of the one at 0150H, so that the original need not be disturbed and in fact, becomes part of the new, stronger CONVERT. You will remember that when we used this subroutine before, we combined it with some instructions for loading and fetching the contents of the accumulator from location 0050H. This was done so that the conversion subroutine would not permanently destroy the original contents of the accumulator while the conversion was executed. We will use a similar technique now, except that eight new memory locations will be required, one for each of the display buffer locations, 0200H through 0207H, that our display subroutine uses to drive the LED displays. We will term these new locations display registers and place them at locations 0208H-020FH.

Display Digit	Display Buffer	Display Register
0	0200H	0208H
1	0201H	0209H
2	0202H	020AH
3	0203H	020BH
4	0204H	020CH
5	0205H	020DH
6	0206H	020EH
7	0207H	020FH

Our new, super version of CONVERT will operate by taking the hexadecimal data out of location 0208H-020FH, converting it to seven-segment code, and replacing the results in locations 0200H-

0207H where the display subroutine can use it. The contents of the display registers at 0208H-020FH should not be disturbed in the process. By placing the display registers right next to the display buffers, we are keeping the data storage portion of the memory at one place. Of course, the data registers could have been put at any place in the memory, but it is considered bad programming practice to spread locations used only for data storage all around. This is because data locations tend to break up the memory, so that programs will have to have jumps in them to allow the program to jump around the storage locations used for data. This is both awkward and a waste of memory since all of those jumps require three locations each. By keeping all of the data storage registers at one place in the memory, this jumping about is minimized. A very good place for keeping data is at the very low order memory locations, 0037H and less. However we have some reasons for not wanting to use those locations just yet; we'll explain why in a few moments.

Since the last location used by the original CONVERT subroutine is at location 0156H, we'll start the new subroutine, CONVERT, at the next highest location 0157H. The program is simplicity itself. The data at location 020FH is fetched with a LDA instruction, converted by CALLing the conversion subroutine at location 0150H, and storing the results, represented by the contents of the accumulator, at location 0207H. This process is repeated for each of the other display buffers and registers. Study the following program and then enter it in your system memory using the DCM mode.

0157	3A	LDA	020FH	;Load the accumulator with the data
0158	0F			;in the display register at 020FH.
0159	02			;
015A	CD	CALL	CONV	;Convert contents of accumulator.
015B	50			;
015C	01			;

015D	32	STA	0207H	;Store converted data in the display
015E	07			;buffer at 0207H.
015F	02			;
0160	3A	LDA	020EH	;Load the accumulator with the data
0161	0E			;in the display register at 020EH.
0162	02			;
0163	CD	CALL	CONV	;Convert contents of accumulator.
0164	50			;
0165	01			;
0166	32	STA	0206H	;Store converted data in the display
0167	06			;buffer at 0206H.
0168	02			;
0169	3A	LDA	020DH	;Load the accumulator with the data
016A	0D			;in the display register at 020DH.
016B	02			;
016C	CD	CALL	CONV	;Convert contents of accumulator.
016D	50			;
016E	01			;
016F	32	STA	0205H	;Store converted data in the display
0170	05			;buffer at 0205H.
0171	02			;
0172	3A	LDA	020CH	;Load the accumulator with the data
0173	0C			;in the display register at 020CH.
0174	02			;
0175	CD	CALL	CONV	;Convert contents of accumulator.
0176	50			;
0177	01			;
0178	32	STA	0204H	;Store converted data in the display
0179	04			;buffer at 0204H.
017A	02			;
017B	3A	LDA	020BH	;Load the accumulator with the data

017C	0B			;in the display register at 020BH.
017D	02			;
017E	CD	CALL CONV		;Convert the contents of accumulator.
017F	50			;
0180	01			;
0181	32	STA	0203H	;Store converted data in the display
0182	03			buffer at 0203H.
0183	02			;
0184	3A	LDA	020AH	;Load the accumulator with the data
0185	0A			in the display register at 020AH.
0186	02			;
0187	CD	CALL CONV		;Convert the contents of accumulator.
0188	50			;
0189	01			;
018A	32	STA	0202H	;Store converted data in the display
018B	02			buffer at 0202H.
018C	02			;
018D	3A	LDA	0209H	;Load the accumulator with the data
018E	09			in the display register at 0209H.
018F	02			;
0190	CD	CALL CONV		;Convert the contents of accumulator.
0191	50			;
0192	01			;
0193	32	STA	0201H	;Store converted data in the
0194	01			display buffer at 0201H.
0195	02			;
0196	3A	LDA	0208H	;Load the accumulator with the data
0197	08			in the display register at 0208H.
0198	02			;
0199	CD	CALL CONV		;Convert the contents of accumulator.
019A	50			;
019B	01			;

```

019C    32    STA    0200H    ;Store converted data in the
019D    00                                ;display buffer at 0200H.
019E    02                                ;
019F    C9    RET                                ;Return to calling program.

```

With the subroutines now in place we should be able to display data in the displays by simply loading it into locations 0208H-020FH and then CALLing CONVERT and DISPLAY. We have done this in the simple program below starting at location 0100H. Load this program and try it.

```

0100    CD    CALL CONVERT    ;Convert data in display registers
0101    57                                ;to seven-segment code in the display
0102    01                                ;buffers.
0103    CD    CALL DISPLAY    ;Display contents of buffers.
0104    50                                ;
0105    02                                ;
0106    C3    JMP     0103H    ;Complete the loop.
0107    03                                ;
0108    01                                ;

```

If you press **CLR** and **EXC** now, you should see all of the displays light, except once again, they will contain garbage. This is because we did not load the display registers with hex digits 0-F. Anything other than one of these sixteen values will be converted into a code that we will not recognize. You can prove this to yourself very quickly by using the DCM mode to manually load the display registers with real data. Load the following:

```

0208    00
0209    01
020A    02
020B    03
020C    04

```

020D	05
020E	06
020F	07

Try the program again.

Press

See Displayed

CLR EXC

76543210

Let's try a simple program using the CONVERT and DISPLAY subroutines that we have developed in the preceding pages. This will be a simple program to cause the display to count. True, we have already done this once, but that was a very confusing program what with all of the delay subroutines and the storing and retrieving of data. We will also introduce a new instruction at this time, INR M, which causes the indirectly addressed memory location pointed-to by the H/L register pair to be incremented. This can be very useful since the data need not be brought into the working registers of the CPU in order to be incremented.

We begin the program at location 0100H by loading the accumulator with 00H so that the counting action will begin at zero. This is followed with a STA 0208H to load the right-most display register with the contents of the accumulator, 00H. CALLing CONVERT causes the contents of all eight of the display registers to be converted and loaded into the eight corresponding display buffers. It will be necessary to set up a timing function for the DISPLAY subroutine. This is because the computer passes through DISPLAY subroutine. This is because the computer passes through DISPLAY so fast, the LEDs are only lighted for a very short time and are not visible. Instead we will cause this to be part of a loop that is passed 256 times so the digits will be visible. The loop is implemented by loading the C register with FFH, which is 256 in decimal, and then CALLing DISPLAY. Immediately after program execution returns from the DISPLAY subroutine, the C register is decremented and then tested with a JNZ instruction. If C still contains a

non-zero number, control passes back to DISPLAY. After the subroutine has been passed FFH times, the test fails and program control falls through to an LXI H,0208H and INR M instructions which cause the count in display register 0208H to be incremented. A JMP back to CONVERT completes the loop. See Fig. 6-8. Load the program below into the system.

0100	3E	MIV A,00H	;Load the accumulator with 00H to
0101	00		;start the counting action at zero.
0102	32	STA 0208H	;Initialize display register by
0103	08		;loading zero.
0104	02		;
0105	CD	CALL CONVERT	;Convert contents of display registers
0106	57		;and load into display buffers.
0107	01		;
0108	0E	MVI C,FFH	;Initialize loop counter.
0109	FF		;
010A	CD	CALL DISPLAY	;Display contents of display buffers.
010B	50		;
010C	02		;
010D	0D	DCR C	;Decrement loop counter.
010E	C2	JNZ 010AH	;Test loop counter. Loop until C=0.
010F	0A		;
0110	01		;
0111	21	LXI H,0208H	;Point H/L to display buffer.
0112	08		;
0113	02		;
0114	34	INR M	;Increment count in display buffer.
0115	C3	JMP 0105H	;Complete loop.
0116	05		;
0117	01		;

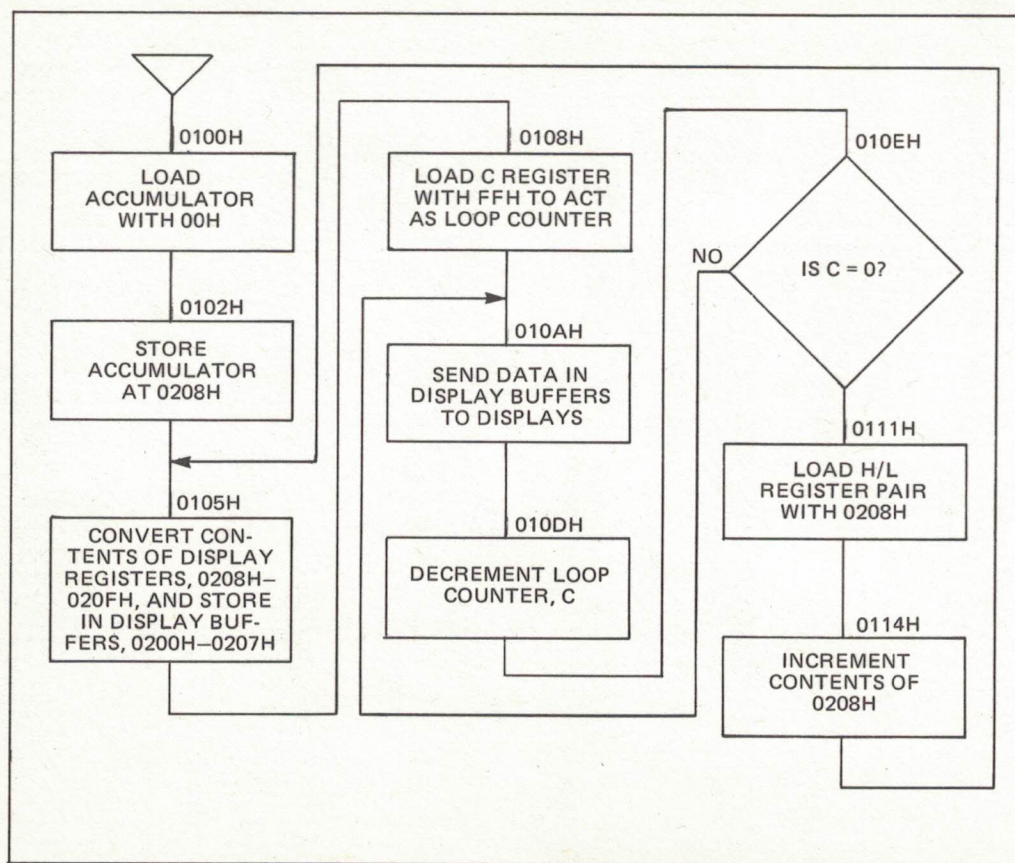


Fig. 6-8 Flow diagram of program to make computer count.

You can execute the program by pressing **CLR** and then **EXC** . The effect will be similar to before except that now all of the displays will be lighted, although only the right-most will be counting. By this point you should be aware of several things. First, our programs are starting to get very long, and there isn't much memory left to do useful things. Second, this is compounded by the fact that our CONVERT, DISPLAY and DELAY subroutines are using up all of the working registers. The accumulator is used in CONV, CONVERT and DISPLAY. The B register is used in DELAY I, the C register as the loop counter in the main program, the D register in the DISPLAY subroutine, and L and H are both used in CONV. Only the E register remains as yet, unused. The lack of memory and unused working registers is becoming a serious bind since there isn't much room left for working program. We can get around the lack of registers by storing their contents in data storage locations in memory. However, this is very inefficient since a STA instruction requires three locations, and the LDA instruction to reload it another three. Together with the location required to store the data itself, seven locations are required to save the contents of one of the working registers. This only compounds the problem of not enough memory.

The Stack Memory. We will now explore a technique for saving the contents of the working registers in the memory without using up seven locations in the process. With the working registers free we can make our subroutine more efficient which will drastically shorten them. That will, in turn, free up more of the memory for working programs. The technique involves a concept called the stack memory. We have worked with this already when we saved the return address during the execution of a CALL instruction. You will remember that when a CALL is executed, the contents of the program counter are loaded into the locations pointed to by the stack pointer. Thus if the stack pointer is set at 0100H, a CALL will cause the program counter contents to be loaded into locations 00FFH and 00FEH. During this process the stack pointer will be decremented twice so that it now indicates 00FEH. When the end of the subroutine is reached and a RET instruction executed, the contents of 00FFH and 00FEH are loaded back into the program counter so that program execution picks up where it left off. At the same time the stack pointer is incremented twice so that it once again contains 0100H.

We can use a similar technique for storing the contents of the working registers. The area of memory pointed-to by the stack pointer and those locations below it is referred to as the stack memory. This is a variable area since the stack grows downward as CALLs load return addresses into the memory, and shrinks back upward as RET instructions fetch the addresses back to the program counter. The new technique saves the working registers in the stack by a set of instructions. Each PUSH saves two of the registers at a time. Thus PUSH H saves both the H register and the L register. PUSH D saves the D and E registers; PUSH B saves the B and C registers. PUSH PSW operates a little differently. It serves to save the contents of the accumulator and all of the flags at the same time. Altogether there are four PUSH instructions; each occupies one memory location. To restore the contents of the working registers, there are four POP instructions, one for each of the PUSH instructions. Executing a POP B causes the former contents of the B and C registers to be reloaded into those registers. Since each PUSH and POP occupies just one memory location, we can save and later restore a working register at an average memory cost of only memory location. Sound too good to be true? Well there are a few rules to follow when you use these instructions, but they're pretty straightforward. The simplest way to illustrate the use of PUSH and POP to save registers is with a demonstration program. Load the following program into the memory:

0100	3E	MVI A,00H	;Load the accumulator with 00H.
0101	00		;
0102	06	MVI B,01H	;Load the B register with 01H.
0103	01		;
0104	0E	MVI C,02H	;Load the C register with 02H.
0105	02		;
0106	16	MVI D,03H	;Load the D register with 03H.
0107	03		;
0108	1E	MVI E,04H	;Load the E register with 04H.
0109	04		;
010A	26	MVI H,05H	;Load the H register with 05H.

010B	05		;
010C	2E	MVI L,06H	;Load the L register with 06H.
010D	06		;
010E	C5	PUSH B	;Save B and C registers on stack.
010F	D5	PUSH D	;Save D and E registers on stack.
0110	E5	PUSH H	;Save H and L registers on stack.
0111	F5	PUSH PSW	;Save accum and flags on stack.
0112	3E	MVI A,FFH	;Load FFH into the accumulator.
0113	FF		;
0114	06	MVI B,FFH	;Load the FFH into the B register.
0115	FF		;
0116	0E	MVI C,FFH	;Load FFH into the C register.
0117	FF		;
0118	16	MVI D,FFH	;Load FFH into the D register.
0119	FF		;
011A	1E	MVI E,FFH	;Load FFH into the E register.
011B	FF		;
011C	26	MVI H,FFH	;Load FFH into the H register.
011D	FF		;
011E	2E	MVI L,FFH	;Load FFH into the L register.
011F	FF		;
0120	F1	POP PSW	;Restore the accumulator and flags
0121	E1	POP H	;Restore the H and L registers.
0122	D1	POP D	;Restore the D and E registers.
0123	C1	POP B	;Restore the B and C registers.
0124	FF	RST 7	;Program stoppers.

The program consists of several basic parts. The first loads the registers with known values, 00H in the accumulator, 01H in the B register, 02H in the C register, 03H in the D register, 04H in the E register, 05H in the H register and 06H in the L register. This is done so that the registers contain known data that will be easily recognized as it appears on the stack. The second portion of

the program consists of the PUSH instructions to load the data in the working registers into locations in the stack memory. When this has been completed the registers are free to use for other purposes since the data can always be restored by a series of POPs. To simulate the registers being used for other purposes, we have included a set of MVI FF instructions that load FF into all of the working registers. Finally a series of POP instructions restores the data in the registers. We will now execute the program step by step.

Press	See Displayed
CLR STEP	H0102-06
STEP	H0104-0E
STEP	H0106-16
STEP	H0108-1E

So far we have loaded the accumulator and the B, C, and D registers. We have also STEPPed up to the instruction at location 0108H which is a MVI E instruction, hex code 1EH, that serves to load the E register with the data work 04H. Although this is a new instruction, it operates just like the others in the MVI R series.

Press	See Displayed
STEP	H010A-26
STEP	H010C-2E
STEP	H010E-35

The instructions at locations 010AH and 010CH are additional instructions in the MVI block. MVI H serves to load the H register with data. MVI L operates in the same way. Now, although 010E-C5 is displayed, this instruction has not yet been executed; it is the NEXT instruction to be executed. Before we do that, let's take a second and use **DCR** to be sure that our data was really loaded into the registers.

Press

DCR

NXT

NXT

NXT

NXT

NXT

NXT

NXT

NXT

NXT

See Displayed

A-----00

F-----

b-----01

C-----02

d-----03

E-----04

L-----06

H-----05

SP-0100

PC-010E

Write Contents

Our data has been correctly loaded into the registers. Although we did not load the flag register directly with one of the program instructions, we ask you to record the contents of the flag

register since it is part of the saving process implemented by the PUSH PSW instruction. Now, since the registers are going to be saved in the stack memory, we want to examine and record the contents of the memory locations affected by the PUSH instruction.

Press	See Displayed	Write Contents
DCM	U - - - - -	
0 0 F 8 NXT	U 00F8 - -	
NXT	U 00F9 - -	
NXT	U 00FA - -	
NXT	U 00FB - -	
NXT	U 00FC - -	
NXT	U 00FD - -	
NXT	U 00FE - -	
NXT	U 00FF - -	

We are now ready to monitor the action of the first PUSH instruction, the PUSH B at location 010EH. Notice that prior to this instruction the stack pointer contained 0100H. During the execution of each PUSH, the stack pointer will be decremented twice. This causes the registers to be stored in locations 00FEH and 00FFH. Even though only one register is referenced in the PUSH B, the instruction operates on register pairs, so that both the B and C register are saved in the stack memory.

Press	See Displayed
STEP	H010F-d5
DCM	U-----
0 0 F E NXT	U00FE-02
NXT	U00FF-01

The initial portion of the program caused register B to be loaded with 01H and the C register with 02H. The 01H value has now appeared in memory location 00FFH and the 02H value in memory location 00FEH. The PUSH B instruction has apparently caused the contents of the B and C registers to be saved in these two locations. The process by which this happens is as follows. First, the stack pointer is decremented once. It now contains 00FFH, and it is this location that will receive the contents of the first register to be saved. Since each of the PUSH instructions operates with a register pair, the question naturally arises as to which register is saved first. Actually we already have the answer to that question, since we just saw that the 01H value was saved in FFH. This means that the first register of the pair to be saved is the B register. After saving the B register, the stack pointer is decremented a second time so that it now contains 00FEH and the contents of the C register saved at this location. The process of saving the registers does not, in itself, disturb their contents. The only register affected is the stack pointer which is decremented twice for each PUSH instruction. Thus,

Press	See Displayed
DCR NXT NXT	Ub-----01
NXT	Uc-----02
NXT NXT NXT NXT NXT	USP-00FE

Pretty much the same operation occurs when we execute the PUSH D instruction at location 010FH. The stack pointer is decremented once to 00FDH and the contents of the D register saved at that memory location. Then the stack pointer is decremented again to 00FCH and contents of the E register saved at that location.

Press	See Displayed
STEP	H0110-ES
DCM 0 0 F C NXT	n00FC-04
NXT	n00FD-03
DCR NXT NXT NXT NXT	uSP-00FC
NXT NXT NXT NXT	

Again, notice that the high order register of the D/E register pair, that is the D register, is saved at the higher memory location and the E register at the lower memory location. If we STEP once more we will execute the instruction at location 0110H which is a PUSH H instruction. This will decrement the stack pointer to 00FBH and save the H register there, and the decrement the stack pointer again to 00FAH and save the L register.

Press	See Displayed
STEP	H0111-FS
DCM 0 0 F A NXT	n00FA-06
NXT	n00FB-05
DCR NXT NXT NXT NXT	uSP-00FA
NXT NXT NXT NXT	

There is one more PUSH instruction, PUSH PSW at location 0111H. This instruction saves both the accumulator and the flag register in the stack. Although we have seen the flag register every time we have examined the working registers with the DCR key, we have never really discussed it. The CPU keeps track of five things during many of the operations that it performs. For instance, if an operation like decrementing a register produces a zero in that register, the CPU notes this fact and remembers it. That is how we are able to test the contents of a register with a JNZ instruction. This instruction does not look at the register under question, but only at the zero flag which is the CPU's scratchpad where it keeps track of whether the last operation resulted in a zero. Besides the zero flag, there are four others, parity, sign, carry and auxiliary carry. We will discuss these as we encounter them in our programs, but for now we want to consider what happens when we save the working registers on the stack in order to free the status of these flags at the same time the working registers are saved so that we can later recreate the state of the CPU as it existed prior to entering a subroutine. Otherwise it would be very awkward to have to recreate the status flags.

The PSU PSW instruction saves both the accumulator and the flags at the same time. First the stack pointer is decremented and the accumulator saved. Then the stack pointer is decremented again and the flags saved.

Press

STEP

DCM 0 0 F 8 NXT

NXT

DCR NXT NXT NXT NXT
NXT NXT NXT NXT

See Displayed

10112-3E

n00F8-

n00F9-00

uSP-00F8

Write Contents

If you compare the contents of location 00F8H with the contents of the flag register that you recorded earlier, you should find a match. Now that we have saved all eight of the working registers with four instructions requiring only four memory locations, we can use the registers for other programs. That has been simulated by the MVI R,FFH instructions which load FFH into all of the registers (except the flag register which is not easily loaded by direct instructions).

Press				See Displayed
STEP	STEP	STEP	STEP	H0120-FF1
STEP	STEP	STEP	STEP	
DCR				0A-----FF
NXT				0F-----
NXT				0b-----FF
NXT				0C-----FF
NXT				0d-----FF
NXT				0E-----68
NXT				0L-----FF
NXT				0H-----FF
NXT				0SP-00F8
STEP				H0121-E1

We have now executed the first of the POP instructions, the POP PSW at location 0120H. This causes the contents of the memory location pointed to by the stack pointer to be loaded into the flag register. In our case, this is 00F8H, but the transfer will not be visible since the flag register has not been changed since PUSH PSW saved it. The stack pointer is then incremented and the contents of the next memory location loaded into the accumulator. In our program this should be location 00F9H. The stack pointer is then incremented again. POP PSW therefore is the exact reverse of PUSH PSW. In fact, if we were to follow PUSH PSW immediately with a POP PSW, the state of the computer would be unchanged except for the memory locations that temporarily stored the accumulator and flag register. We can prove this to ourselves by examining the accumulator now. It should have its original contents, 00H, restored from location 00F9H.

Press

See Displayed

DCR

0A-----00

If we **STEP** the next instruction, POP H at 0121H, we cause the contents of the memory location pointed to by the stack pointer, 00FA, loaded into register L. The stack pointer is then incremented, the contents of that memory location loaded into register H and the stack pointer incremented again.

Press

See Displayed

STEP

F0122-d1

DCR	NXT	NXT	NXT
NXT	NXT	NXT	

0L-----06

NXT

0H-----05

NXT

0SP-00FC

The H and L registers have been restored and the stack pointer incremented twice to 00FCH. We now see how the stack memory shrinks upward as POPs are executed. If we STEP again we will execute the POP D instruction at location 0122H. This loads the contents of 00FDH into the E register and the contents of 00FEH into the D register.

Press	See Displayed
STEP	H0123-01
DCR NXT NXT NXT NXT	0d-----03
NXT	0E-----04
NXT NXT NXT	0SP-00FE

Finally we STEP once more to execute the POP B instruction located at 0123H. This causes the C register to be loaded with the contents of location 00FEH and the B register to be loaded with the contents of 00FFH. We have thus managed to restore all of the working registers with just four POP instructions, and they only occupy four memory locations.

Press	See Displayed
STEP	H0124-FF
DCR NXT NXT	0b-----01
NXT	0C-----02
NXT NXT NXT NXT NXT	0SP-0100

The final POP instruction brings the stack pointer back to its original value of 0100H. This is what we mean by a balanced stack. That is, for every PUSH there is a balancing POP, and for every CALL there is a balancing RET. That way every subroutine will be exited with the stack pointer at the same value as when the subroutine was entered. This is considered good programming practice and strict adherence to it will save countless headaches.

Before we leave the subject of the stack, we should point out one thing. The computer does not keep track of what order was used to save the various registers. There is nothing magic about saving the registers with PUSH B, PUSH D, PUSH H, and PUSH PSW in that particular order. Placing PUSH PSW first would work just as well. Also, in many cases there will be no need to save all of the registers upon entering a particular subroutine. Short subroutines may only use one or two of the working registers and in those cases, only the registers to be used should be saved. Keep in mind that the registers are saved in pairs. It would be foolish, for instance, to write a subroutine that used the B register and the D register, since saving these would require both a PUSH B and PUSH D instruction. If instead, the B and C registers are used in the subroutine, only a PUSH B will be required to save the registers.

The most important rule to keep in mind is that whatever order is chosen for the PUSH instructions, the POPs must occur in exactly the opposite order. A moments reflection should prove this to your satisfaction. If the first PUSH is a PUSH B, the B register will be saved in 00FFH and the C register in 00FEH. If the next stack operation is a POP D, the contents of 00FEH will be loaded into the D register and the contents of 00FEH into the E register. This is exactly the opposite of what was intended and will undoubtedly cause errors. Occasionally a programmer will do these kind of switch operations on purpose to simplify subroutines, but great care must be used.

This is easily demonstrated by switching the POP B and POP D instructions in our demo program. This will reverse the order in which the register pairs are restored.

Press	See Displayed
DCM 0 1 2 2 NXT	n0122-d1
C 1	n0122-C1
NXT	n0123-C1
D 1	n0123-d1
NXT	n0124-FF
CLR EXC	F0125-00
DCR NXT NXT	u6-----03
NXT	uC-----04
NXT	ud-----01
NXT	uE-----02

By exchanging the order in which the two POP instructions appear, we have reversed the order in which the registers are reloaded. Now, let's try the new stack techniques to see if we can streamline CONVERT.

Final Conversion Routine. Now that we have the ability to save the working registers on the stack memory, we should be able to drastically shorten the CONVERT subroutine located at 0150H. This is because being able to use all of the working registers in the subroutine allows us to indirectly address the memory locations, and, as we shall see shortly, this will result in the savings of many memory locations. The subroutine is illustrated in the flow diagram of Fig. 6-9.

We begin the new version of the CONVERT subroutine by saving all of the working registers on the stack with a series of four PUSH instructions. Although this could be done in the main program prior to calling the subroutine, this is an inefficient approach. After all, the whole purpose of setting a program segment up as a subroutine is to be able to CALL it from many places in the main program. If the saving and restoring of the registers were to be done in the main program before and after the subroutine, the process would have to be repeated each and every time the subroutine were CALLED. By placing the PUSH and POP instructions within the subroutine the instructions will only appear in memory once.

After the process of saving the registers has been completed, they are free to use for our own purposes. We will load the B/C register pair with 0207H, the address of the high-order display buffer. This is done with an LXI B,0207H instruction. We then point the D/E pair to 020FH, the address of the high-order display register, with an LXI D,020FH instruction. This is a new instruction to be added to our repertoire, but operates in the same way as LXI B and LXI H that we have used previously. The next step is to use an LXI H to load the starting address of the seven-segment look-up table, 8282H.

We have now pointed the three register pairs at three different areas of memory, and are ready to use them to indirectly address those locations. The B/C pair is pointing to the display buffers, the D/E pair is pointing to the display registers, and the H/L pair is pointing to the look-up table.

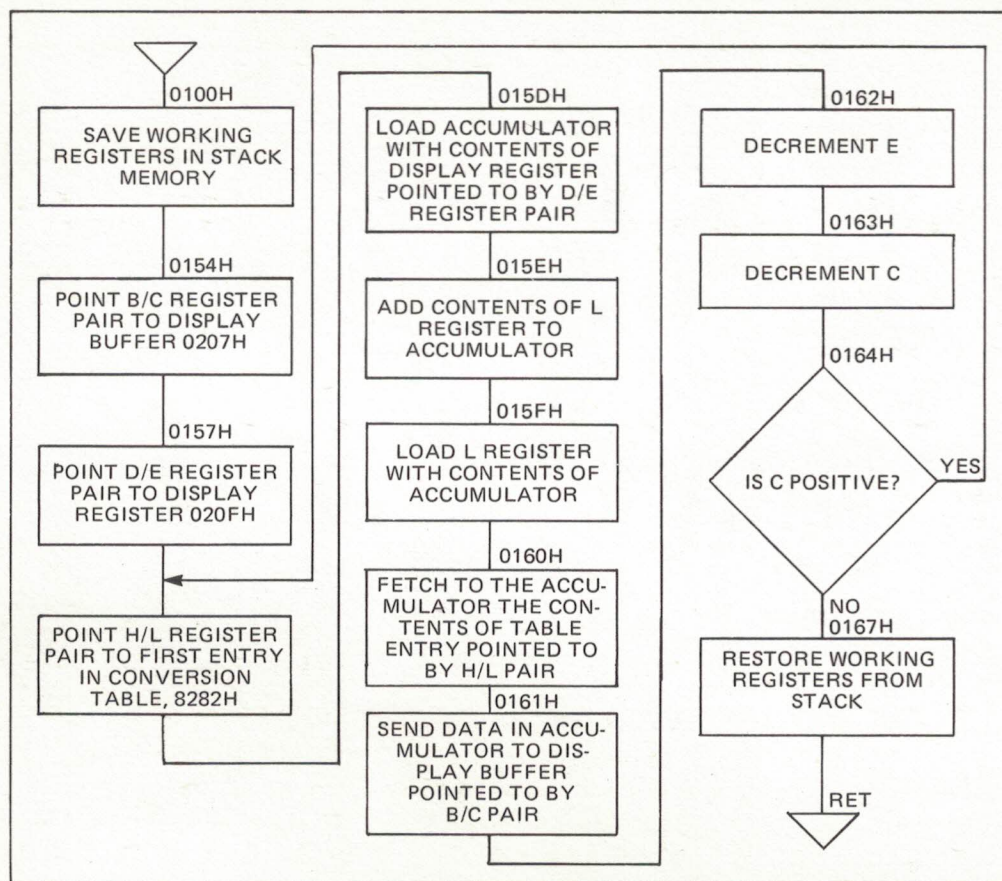


Fig. 6-9. Flow diagram of the final CONVERT subroutine.

An LDAX D instruction loads the contents of the memory location pointed to by D/E into the accumulator. This is another new instruction, but it operates in the same fashion as the STAX D instruction already discussed. That instruction, you will remember, sent the contents of the accumulator to the memory location specified by the D/E register pair. LDAX D is the exact reverse, that is, it fetches to the accumulator the contents of the location specified by D/E. When this instruction is executed in our program, it will serve to load the accumulator with the contents of the memory location we have selected to act as our display register. Thus the accumulator contains the first of the hex digits to be converted into seven-segment code. The contents of the L register are added to the accumulator and the results stored in the accumulator. Since H/L were initially pointed to the first entry of the look-up table, the accumulator now contains the low-order two digits of the address corresponding to the appropriate entry in the table. MOV L,A loads the new value back into the L register so that H/L now point to the correct spot in the table. MOV A,M fetches the correct code from the table to the accumulator. This is the second indirect memory access. The third is a STAX B instruction that sends the code now residing in the accumulator to the display buffer pointed to by the B/C register pair. This completes the conversion process for display register 020F and display buffer 0207H.

We will now go on to the other buffer/register pairs. Here we will see the real savings in program memory locations. Our last version of CONVERT went through the same program eight times in order to perform the conversion on each of the buffer/register pairs. With the working registers pointing at the buffers and registers, the conversion process on the remaining pairs will be considerably simpler in our new version of the program. A DCR E instruction causes the D/E pair to point to the display buffer 0206H. We can now make our earlier instructions do double duty by jumping back to them and performing the conversion again. In fact, this same process will cause the computer to convert all of the data in the display registers to the correct code in the display buffers. This process cannot go on indefinitely, however since the last buffer is the one at location 0200H, and must terminate the subroutine. The natural tendency is to test for the last

buffer by placing a JNZ instruction after the the DCR C. Remember, even though the buffers addressed by the B/C register pair, the C register is the only one that is decremented. Its contents range from 07H to 00H as the buffers 0207H to 0200H are addressed. Unfortunately, a JNZ test will cause the test to fail on 00H and the last buffer at 0200H will be skipped.

The solution to our problem lies in the way the computer handles signed numbers. After all, every pocket calculator has the ability to work with both positive and negative numbers, and it would be a sorry computer that couldn't do at least as well. Actually the electronics in most computers and pocket calculators dodge this issue and leave signed numbers up to the programmer. The user of the pocket calculator is not aware of the problem, but the designer of the calculator had to do a lot of clever programming in order to get those plus and minuses into the system.

Consider for a moment the way the computer handles the zero flag. Certain operations result in the computer storing the fact that the operation did or did not cause a register to contain zero. Many of the instructions do not affect any of the the flags; others set only certain of them. This means that a flag may be set as the result of DCR C producing zero contents in the C register, and that flag remaining valid through several more instructions because these happen to be instructions that do not affect that particular flag. This is not a random event, but one that is carefully controlled by the programmer. In fact, the reason that certain of the instructions do not affect flags that at first glance one would think should be affected, is that certain frequently occurring programming problems are made considerably easier by having the flag remain valid from a previous instruction. The important thing to remember is that the computer sets the flags independently of whether the programmer is going to test them later with some form of a jump or call instruction. After all, at the time the instruction is executed, the computer has no way of looking ahead in the program to see whether the flag is going to be of concern to the programmer or not. So, it sets the flags in accordance with the instructions at the time they are executed.

If we seem to be belaboring this point it is because many new programmers become confused when the subject of the sign flag is brought up. As we have seen, data is represented in the computer with two hexadecimal digits. The numbers represented by these two digits are normally assumed to be positive, which means that the range of numbers that can be expressed is 0 to FFH. However, if we are concerned about expressing both positive and negative numbers we can do so by using the most significant hex digit to carry the sign information. This is done at the price of giving up some of the range of numbers that can be expressed. In this system numbers from 00H to 7FH are considered to be positive and numbers from 80H to FFH are negative. In effect this system changes the range from 00H to FFH where all numbers are positive to a system where the range is -80H to +7FH. How does the computer know which system a data word is being expressed in? It doesn't! None of the instructions that perform addition and subtraction in the computer differ depending upon the signs of the number. But, for purposes of setting the flags, the computer assumes that every number is being expressed in the signed convention. Whether or not this information is used is up to the programmer and the way he utilizes his conditional jump and calls. Most of the time the sign flag is ignored; only in those cases where the programmer is operating with signed numbers will he test the sign flag.

Since numbers in the range of 00H to 7FH are considered to be positive, in the signed numbers system, and 80H to FFH are negative, we can use the sign flag in our program of converting the contents of display registers to codes in the display buffers. Remember, the contents of the C register will decrease from 07H to 00H as the display buffers are driven. These are all positive numbers in the signed numbers system. If we decrement the C register one too many times it will change from 00H to FFH, which is a change from a positive number to a negative number. Our original guess that the conditional jump should be a JNZ was close, but it caused a jump as C changed from 00H to FFH, as desired. JP is another of the class of conditional jumps. It is the conditional jumps, calls and returns that allow the programmer to test the flags. **THESE ARE THE ONLY INSTRUCTIONS THAT UTILIZE THE FLAGS.** By placing a JP 015AH instruction

after DCR C, we cause the program to loop through the conversion until the last display register is converted and the result loaded into the last display buffer. The program is listed below. It requires only 28 memory locations compared to the earlier version's 80 memory locations. Load the program into the system memory.

0150	C5	PUSH B	;Save the working registers in the
0151	D5	PUSH D	;stack memory.
0152	E5	PUSH H	;
0153	F5	PUSH PSW	;
0154	01	LXI B,0207H	;Point B/C to display buffer.
0155	07		;
0156	02		;
0157	11	LXI D,020FH	;Point D/E to the display register.
0158	0F		;
0159	02		;
015A	21	LXI H,8282H	;Point H/L to first entry in seven-
015B	82		;segment table.
015C	82		;
015D	1A	LDAX D	;display register.
015E	85	ADD L	;Add contents of L to accumulator.
015F	6F	MOV L,A	;Move contents of accumulator to L.
0160	7E	MOV A,M	;Fetch table entry to accumulator.
0161	02	STAX B	;Store code in display register.
0162	1D	DCR E	;Next display register.
0163	0D	DCR C	;Next display register.
0164	F2	JP 015AH	;Loop until all registers converted.
0165	5A		;
0166	01		;
0167	F1	POP PSW	;Restore registers from stack.
0168	E1	POP H	;
0169	D1	POP D	;
016A	C1	POP B	;
016B	C9	RET	;

We can prove this program works by placing a short program at 0100H which CALLs CONVERT and DISPLAY. We must first, however, make sure that the display registers contain valid hexadecimal numbers. Load the following data and program into the system.

0208	00		
0209	01		
020A	02		
020B	03		
020C	04		
020D	05		
020E	06		
020F	07		
0100	CD	CALL	CONVERT ;Call the conversion subroutine.
0101	50		;
0102	01		;
0103	CD	CALL	DISPLAY ;Display the contents of the
0104	50		;buffers.
0105	02		;
0106	C3	JMP	0105H ;Complete the loop.
0107	03		;
0108	01		;

Pressing **CLR** and **EXC** should cause the data to be displayed in the LEDs. We have successfully shortened our program drastically and have freed all of the working registers at the same time. In just a moment, we will turn our attentions to the DISPLAY subroutine and see if we can shorten it the same way. First, however, we want to point out there is even an easier way to do the conversion. In the table below we have listed the CONVERT subroutine on the right. On the left are some addresses that we would like you to examine using DCM and record the contents.

Address to Check	Record Contents	Hex Code of our program	Address of our program	Mnemonic
8132H	_____	C5H	0150H	PUSH B
8133H	_____	D5H	0151H	PUSH D
8134	_____	E5H	0152H	PUSH H
8135	_____	F5H	0153H	PUSH PSW
8136	_____	01H	0154H	LXI B,0207H
8137	_____	07H	0155H	
8138	_____	02H	0156H	
8139	_____	11H	0157H	LXI D,020FH
813A	_____	0FH	0158H	
813B	_____	02H	0159H	
813C	_____	21H	015AH	LXI H,8282H
813D	_____	82H	015BH	
813E	_____	82H	015CH	
813F	_____	1AH	015DH	LDAX D
8140	_____	85H	015EH	ADD L
8141	_____	6FH	015FH	MOV L,A
8142	_____	7EH	0160H	MOV A,M
8143	_____	02H	0161H	STAX B
8144	_____	1DH	0162H	DCR E
8145	_____	0DH	0163H	DCR C
8146	_____	F2H	0164H	JP 015AH
8147	_____	5AH	0165H	
8148	_____	01H	0166H	
8149	_____	F1H	0167H	POP PSW
814A	_____	E1H	0168H	POP H
814B	_____	D1H	0169H	POP D
814C	_____	C1H	016AH	POP B
814D	_____	C9H	016BH	RET

Certainly are close, aren't they? In fact, the program at location 8132H is a carbon copy of the one we just wrote and debugged with a few minor address changes. Remember a few pages ago we said that one of the best places to store data was at locations 0000H to 0037H, but for some yet to be stated reason we weren't going to use those locations yet? Well, the reason that we couldn't use those locations is that the ia7301 monitor uses them itself. In fact, it uses them for display buffers and registers in exactly the same way that our program uses locations 0200H to 020FH. This means you have to be very careful when using locations for data storage, since the monitor will wipe out any data in them as it drives the displays. One of the first questions asked by every adventurous student is how come those memory locations don't seem to work when he tries to write into them using the DCM key? The answer, of course, is that the monitor program destroys the data in the very progress of trying to display it. We're going to be using this subroutine a lot in the pages ahead. The address of the CONVERT subroutine in the monitor is 8132H and simply CALLing this address will cause all of the display registers to be converted into seven-segment code in the display buffers. Of course, the display buffers now appear at locations 0000H to 0007H and the display registers at 0008H to 000FH. Since the monitor provides us with a CONVERT subroutine at 8132H, we can eliminate the subroutine that we just wrote altogether.

Geography. Meantime, we have raised a very important question. If there are some low order RAM locations that are used by the monitor, and hence, unusable by us, are there others? It is high time that we settled this once and for all. The answer to the question will take the form of a Memory-I/O map that details the addresses of the various parts of the system. We have already discussed many of these addresses; for example, locations 0100H to 03FFH are RAM memory locations that can be used for program storage. The map appears in Fig. 6-10.

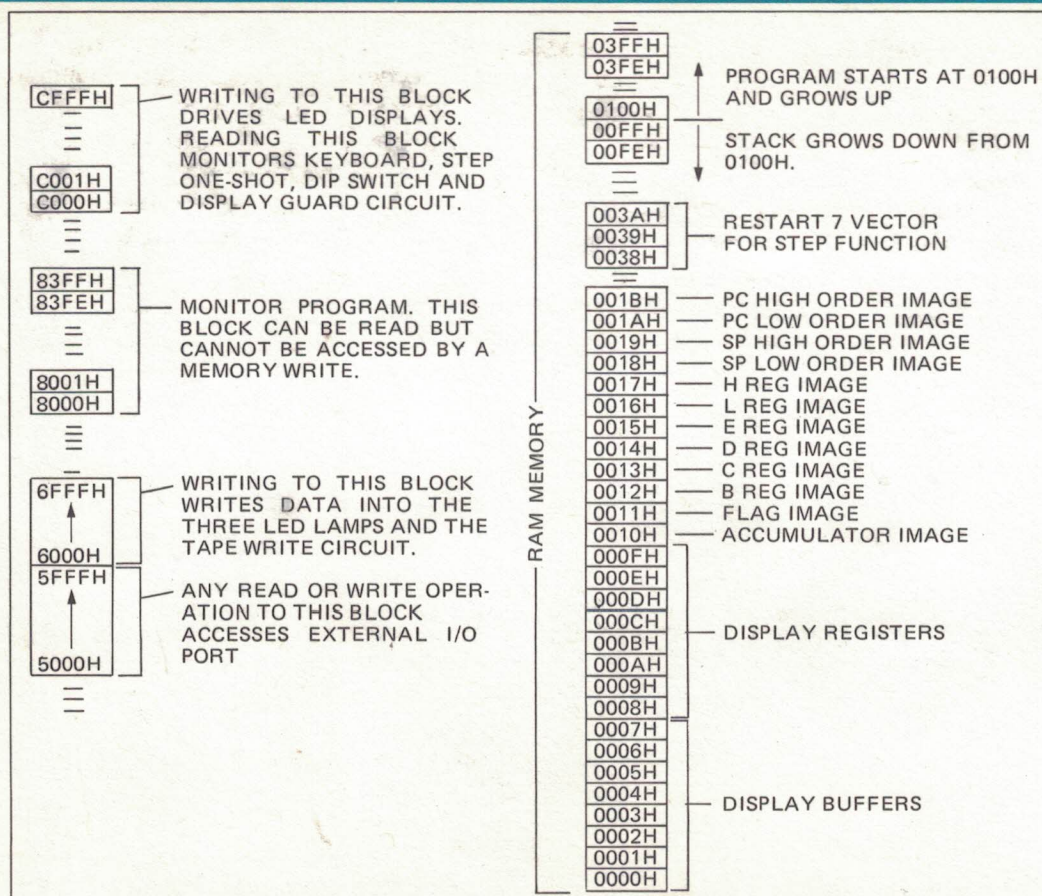


Fig. 6-10. Memory I/O map.

At the very low order RAM locations are the addresses used by the monitor to service the displays. Thus, locations 0000H through 0007H are the display buffers. The contents of these locations must be in the form of seven-segment code if the information on the displays is to be useful. Immediately above these locations are the display registers. Sending data to these eight locations, 0008H through 000FH, and the CALLING CONVERT will cause the contents of the display registers to be loaded into the corresponding display buffers after first being converted to seven-segment code. Keep the addresses of the display registers in your mind; we'll be using them constantly during the remainder of this book. See Fig. 6-11.

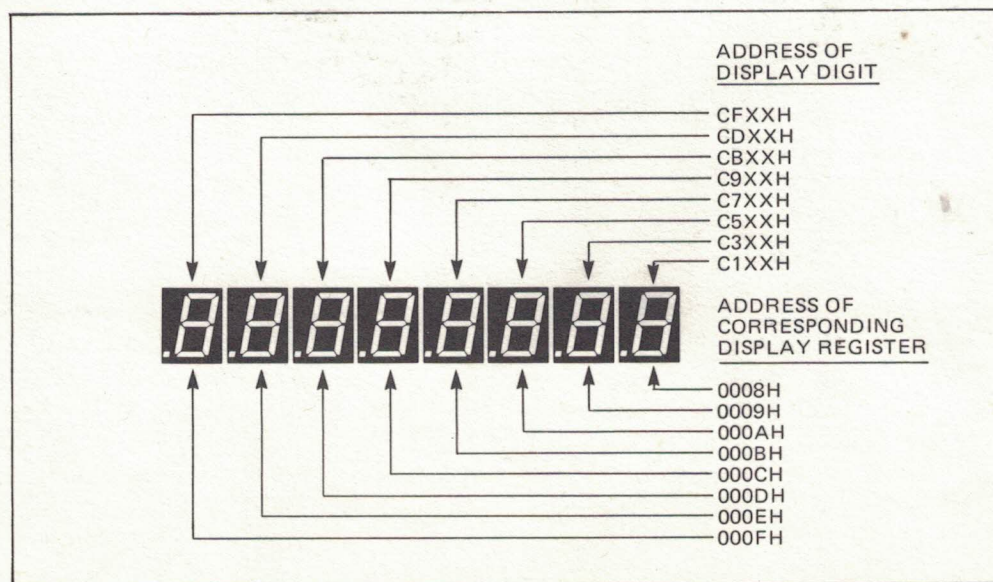


Fig. 6-11. Map of displays and corresponding display registers.

Immediately above the display registers at 0008H to 000FH are a set of memory locations that the computer uses to store the images of the working registers within the CPU. If you have been following this course carefully, it has probably occurred to you that using the DCR key to examine the accumulator, for instance, is a strange process. As we have seen, the driving of the LED displays is a complex process that requires the use of all of the working registers including the accumulator. How then, can we look at the contents of the accumulator, since the very process of displaying that register uses the register itself. Watching the LEDs display the content of the accumulator implies that the accumulator is a static, unchanging number. Yet at the very moment that we are looking at the displays, the accumulator is changing perhaps ten thousand times a second or even more.

The answer is that pressing **DCR** does not display the contents of the accumulator. It displays the contents of memory locations 0010H which we have labeled the Accumulator Image. If this is true, it is certainly within the bounds of reason since the contents of that location are not changing continuously, whereas the contents of the accumulator are. Why then, do we think we are seeing the contents of the accumulator when we press **DCR** ? The answer to that lies in the way that **STEP** and **EXC** work.

When we press the **EXC** key, the computer jumps to a place in the monitor program called EXECUTE. This program fetches from the low-order RAM locations the data in the locations labeled accumulator image, flag image, B register image, etc., and loads the data into the correct registers within the CPU. Portions of this program get pretty tricky; loading the program counter with the contents of the image locations dedicated to that register without changing the value of the program counter in the process requires some thought. When it is done correctly, however, the computer completes the EXECUTE program with all of the registers correctly loaded, and a jump is executed to the value in the program counter. This causes the computer to begin executing your program.

When **STEP** is pressed the reverse process is begun. Each of the working registers is saved in the memory location set aside for it. When this process is completed, the computer displays the contents of the program counter image locations and the contents of the memory location pointed to by the program counter. Pressing **STEP** again at this time, will cause the registers to be reloaded back into the image locations.

Thus the memory locations 0010H through 001BH operate as temporary holding registers that store the contents of the working registers as they were at a particular instant of time. These locations become a sort of transient scrapbook of data, changing each time **STEP** or **DCR** is used. This approach vastly simplifies the process of displaying and changing the contents of the working registers.

Above location 001BH, the last of the image registers, the memory is free until location 0038H is reached. This area is too small to be very useful for programs, but it is ideal for storing data. Locations 0038H, 0039H and 003AH are used to store a JMP instruction. The computer is electronically sent to this location whenever the **STEP** key is pressed. This is not a particularly useful place for the computer to go, but it is a function of the integrated circuits that make up the ia7301 and most other computers. Since we want the **STEP** key to result in the computer entering the STEP routines in the monitor, it is necessary to insert a JMP instruction at these locations. Thus when the **STEP** key is pressed the computer control is transferred to 0038H where it encounters a JMP back into the monitor program.

Above locations 003AH, the memory is once again free until location 0100H is reached. This is the normal starting point at which you will begin your program. That program is begun at 0100H and progresses upward to the last memory location, 03FFH. Although we say that the memory is free between 003AH and 0100H, some care need be exercised. This is because the computer uses the memory immediately below 0100H to build the stack that it uses to save the

contents of registers and return addresses when doing subroutine CALLs. It would be nice if we could place a dividing line somewhere in this area of the memory, but the fact is that the stack is a dynamic thing that changes size as the program is executed. This means that if you are planning to use this area of the memory, you should start at 003BH and work upward, restricting your use of the higher locations as much as possible. It is pretty hard to imagine any program capable of running on the ia7301 that would extend the stack below location 0099H, so the stack would be limited to locations 00A0H to 00FFH. If there is any question about how far down the stack reaches in a given program, the best thing to do is use DCM to load 00H into the locations 003AH to 00FFH, and then **STEP** or **EXC** the program in question. The memory can then be reexamined and it will be immediately obvious how many locations have been distributed by the stack. In any event, it is probably wise for the beginning programmer to avoid locations 0099H to 00FFH until he becomes experienced in the ways of the stack.

Although the RAM memory ends at location 03FFH, it appears to go higher. This is because the electronic circuits that send out the addresses to the memory do not make a distinction between locations 0000H and 0400H, 0001H and 0401H, 0002H and 0402H, etc. This anomaly extends up to 0FFFH. While it appears to be possible to address and change locations in the range of 0400H to 0FFFH, although all that is really happening is the locations in the range of 0000H to 03FFH are being accessed.

At location 5XXXH is the external I/O port (the X's here indicate that any numbers may appear in these places — the port will be accessed by any operation to a location beginning with 5 in the MSD place). If we do a memory read operation to location 5000H, we read the input port. You can demonstrate this with the **DCM** key by keying in address 5000H and pressing **NXT**. The displays should show the contents of this location as FFH, since the computer reads either a high voltage or an open connection as a logic 1. Normal memory read operations are LDA, MOV A,M

and LDAX D, to name a few. Doing a memory write to this address with a STA, MOV A,M or STAX D will send data to the output port. When this happens, the data sent to the port will be inverted so that, for example, sending FFH to the port will cause 00H to actually appear there. This inversion only takes place when sending data to the output port; the input port receives straight, uninverted data.

The external I/O port is so termed because it is connected to the outside world. This is the only one of the I/O ports in the ia7301 that can interface electrically to other external circuits. Although we will not go into the interface requirements in this book, the actual circuits are very straightforward. Integrated circuits can be purchased which can connect directly to the output port and are capable of driving an ampere or more. Small displays can be connected to the port so that AC can be controlled for driving heavy loads.

As far as programming goes, the external I/O port is treated in exactly the same fashion as the ports that we have already dealt with. Some loads will require that their driving signals be pulsed with some very exacting timing requirements. For instance, true of stepper motors that are found in many types of printer mechanisms. The techniques for doing this are the same as those for driving the multiplexed LED displays.

Writing data to the 6XXXH block causes the three discrete LEDs to be lighted. This block also contains the tape recorder write circuitry that we will discuss in the next chapter. Locations 8000H to 83FFH are the PROMs that contain the system monitor. The CXXXH block of addresses is used in two ways. Writing to this block drives the LED displays. Reading this block reads a special register which contains data from the keyboard, the STEP circuits, the DIP switch, and the display guard timing circuit. This will be discussed in more detail later.

The Digital Clock Program. So far, we have confined our programming to problems that were not directly useful. Oh, it's important to be able to turn a load on and off, and the timing and address problems associated with putting numbers onto a set of LEDs are interesting, but this is just not the kind of thing you'd call a neighbor over to see. We're going to remedy that right now by tackling a real problem — a digital clock. We'll make use of all the techniques and instructions we've learned so far, and the result will be something you can be proud of.

Every clock needs some sort of timekeeping base, that is a periodic event whose occurrence is so regular that it becomes the basis of the timekeeping instrument. Old clocks used pendulums for this; modern watches use a balance wheel. The new electronic watches use a quartz crystal that oscillates with an exact, never varying frequency. The clock or watch then counts these oscillations and uses this count to derive the time. It so happens that the ia7301 uses one of these quartz crystals oscillating at 8,000,000 times a second to generate the systems clock needed by the integrated circuits that make up the computer. This means that the operations of the ia7301 are carried out with fantastic accuracy in terms of the time required to perform them. We will base our clock program on this fact; the result will be accuracy in the range of a few seconds a month.

The first step will be to write a DELAY program that will become our basic timekeeping interval. This will be done by just having the computer execute a little loop until the desired delay has been achieved. This program has a second use which is to simulate the guard circuit timing out for the displays. Why don't you go ahead and write this subroutine starting at location 0300H. The timing range is such that only one register is needed for the decrement and loop operations, and while you can use any register you wish, we used the B register. Since we're going to need all of the registers during this clock program, be sure to save it on the stack and later restore it so we won't tie the register up. Start with 4FH as the number of passes through the loop. Our version of the program occupied only locations 0300H through and including 0308H. Can you match it? You'll find the correct program in Fig. 6-12.

DELAY SUBROUTINE

Address		Write Contents Here
0300H	_____	_____
0301H	_____	_____
0302H	_____	_____
0303H	_____	_____
0304H	_____	_____
0305H	_____	_____
0306H	_____	_____
0307H	_____	_____
0308H	_____	_____

Display One-Pass. Obviously we'll need some way of displaying the time on the LEDs. Of course, we already wrote a display program earlier in this chapter, but that was not very elegant and required over 60 memory locations. It worked by using an LDA instruction to load the accumulator with the contents of the display buffer, and then a STAX D instruction to send those contents to the display. Since there are eight displays, the program had to be repeated eight times. We are now going to streamline that program by making more extensive use of indirect addressing and using a loop to repeat the program eight times instead of simply repeating the coding. We will place this program right next to the DELAY subroutine we just wrote. The first location will, therefore, be at 0309H. The flow diagram is at Fig. 6-13.

0309	F5	PUSH	PSW	;Save working registers
030A	D5	PUSH	D	;
030B	E5	PUSH	H	;

DELAY			
0300	C5	PUSH B	;This subroutine
0301	06	MVI B, 4FH	;Causes a short
0302	4F		;Delay. It uses
0303	05	DCR B	;Register B
0304	C2	JNZ 0303H	;No registers
0305	03		;Are disturbed
0306	03		;During this
0307	C1	POP B	;Subroutine
0308	C9	RET	;

Fig. 6-12. The program listing for DELAY.

030C	16	MVI D,CFH	;Point D/E to most significant display
030D	CF		;digit, CFXXH.
030E	21	LXI H,0007H	;Point H/L to display buffer No. 7.
030F	07		;
0310	00		;
0311	7E	MOV A,M	;Load accumulator with contents of
0312	12	STAX D	;display buffer. Send contents of
0313	CD	CALL DELAY (0300H)	;accumulator to display. Delay to
0314	00		;simulate guard circuit timing out.
0315	03		;
0316	15	DCR D	;Point D/E to next display digit.
0317	15	DCR D	;
0318	2D	DCR L	;Point H/L to next display buffer.
0319	F2	JP 0311H	;If not last buffer, repeat.
031A	11		;
031B	03		;
031C	E1	POP H	;Restore working registers.
031D	D1	POP D	;
031E	F1	POP PSW	;
031F	C9	RET	;Return to calling program.

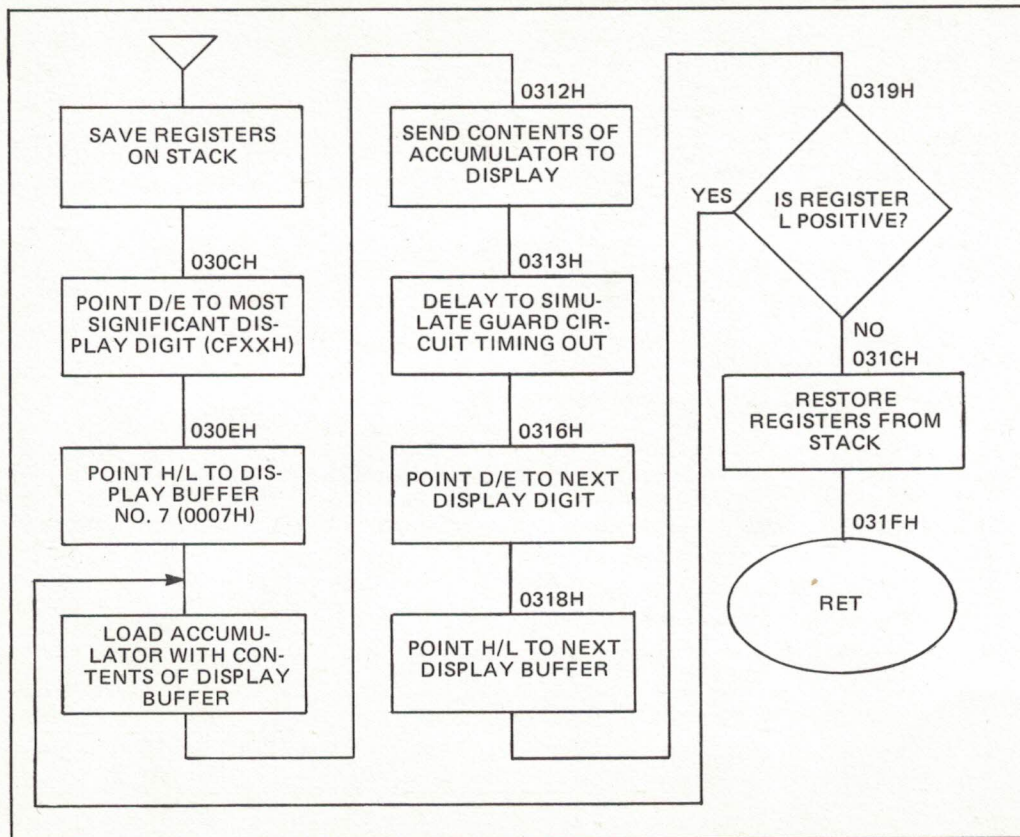


Fig. 6-13. Flow diagram for DISPLAY ONE - PASS subroutine.

This program should now appear very straightforward to you. It operates by saving all of the working registers on the stack. D/E are then pointed to the most significant of the LED display digits. You will remember that these digits are addressed by CFXXH, CDXXH, CBXXH, C9XXH, ect. The X's imply that the circuits don't care what the two least significant hex digits are; they will be addressed correctly regardless. This means that it is not necessary to load the E register, and a MVI D,CFH instruction is sufficient to point the pair to the correct display digit. The H/L register pair is then pointed to the corresponding display buffer in memory, 0007H. These memory circuits are not as easy to please as the LEDs were; both H and L must be set up. This is done with an LXI H, 0007H instruction. Now that both pairs are pointing to the correct areas, the data transfer is easy. MOV A,M loads the accumulator with the contents of the display buffer. STAX D causes this data to be sent to the LED. CALLing DELAY simulates the action of the guard circuit timing out.

This completes the driving of the first digit. To drive the remainder, we decrement D/E twice so that it points to the next LED. A DCR L instruction causes the H/L pair to point to the next display buffer at location 0006H. Although we have not used the DCR L instruction before, it operates exactly as DCR B, DCR C, DCR D and DCR E that are already in our repertoire. It is now necessary to test to see if the registers are pointing to valid addresses. By this we mean that after driving eight displays, the decrementing process will cause D/E to point to BFXXH which is not a valid display address. Likewise decrementing L at this point will cause H/L to point to FFFFH which certainly does not correspond to any of the display buffers. Fortunately we already have a trick for terminating the subroutine at this point. We'll use the same method as for CONVERT. This last decrementing operation changed the contents of L from a positive number to a negative number, and in the process, set the sign flag. We can test that flag with a JP instruction, Jump if Positive, which causes the program to loop back upon itself if the registers are proven to be still pointing to valid addresses. All that remains is to restore the working registers and return to the CALLing program.

One-Second. Our clock is going to change the displays once a second, so we're going to have to combine our programs so that we end up with a one-second subroutine during which the displays are active. This is simplicity itself. The flow diagram is in Fig. 6-14.

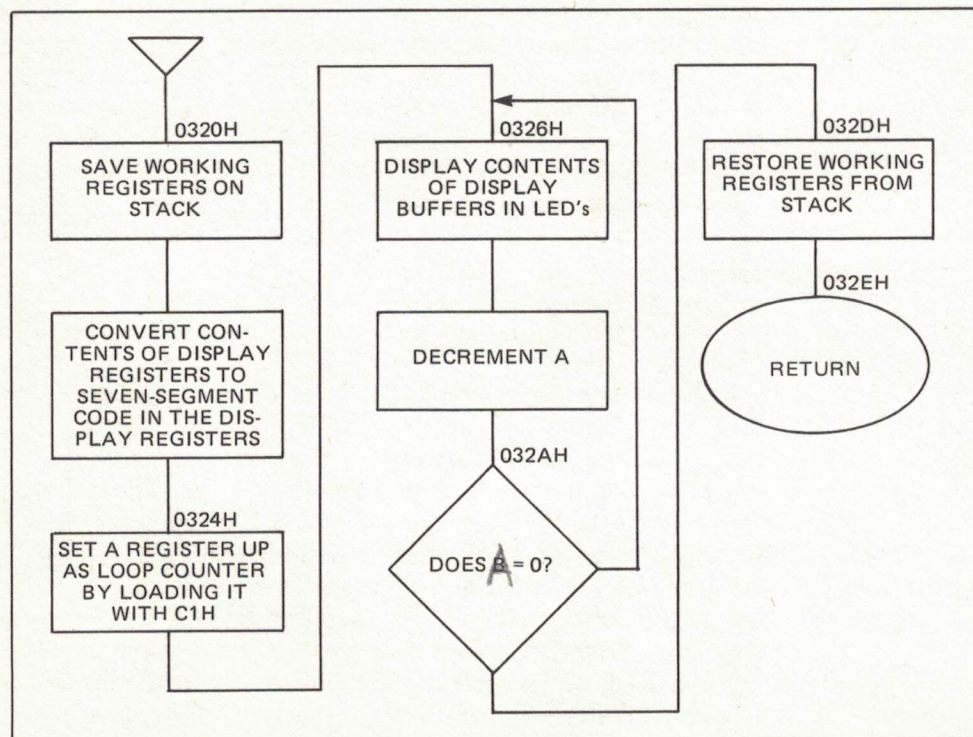


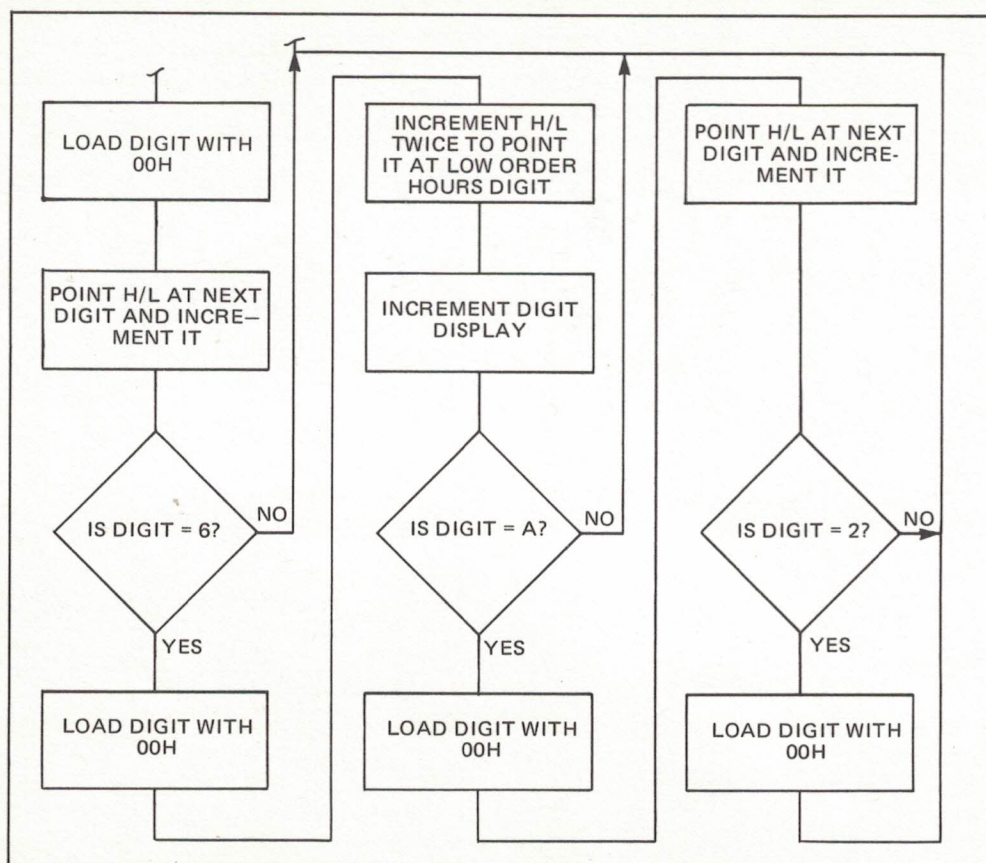
Fig. 6-14. Flow diagram of ONE — SECOND subroutine.

To implement this ONE-SECOND subroutine all that is necessary is to perform DISPLAY ONE PASSes until the second is up. We can do this by setting up a loop around the display subroutine. This is done by using the accumulator as a loop counter, decrementing it on each pass, and then testing it for zero contents.

ONE-SECOND

0320	F5	PUSH PSW	;Save accumulator on the stack.
0321	CD	CALL CONVERT (8132)	;Convert display registers to
0322	32		;display buffers.
0323	81		;
0324	3E	MVI A,C1H	;Load accumulator with timing constant.
0325	C1		;
0326	CD	CALL DISPLAY ONE-PASS (0309H)	;Display contents of
0327	09		;display buffers in LEDs.
0328	03		;
0329	3D	DCR A	;Decrement loop counter.
032A	C2	JNZ 0326H	;Test for zero contents.
032B	26		;
032C	03		;
032D	F1	POP PSW	;Restore accumulator from stack.
032E	C9	RET	;Return to calling program.

The only new instruction in this subroutine is DCR A, Decrement Accumulator, which operates in the same fashion as DCR B, DCR C, etc. The ONE-SECOND subroutine combines CONVERT and DISPLAY ONE-PASS so that all of our display function are taken care of automatically by CALLing this subroutine. The subroutine is exited exactly one second later, after displaying the contents of the display registers during that time. None of the working registers are disturbed in the process, having been saved on the stack at the appropriate times.



The Clock. With these basic subroutines in the 300H block of memory, we're all set to go ahead and write the main program. This will appear at 0100H to take advantage of the automatic CLR feature of the computer. The program operates by loading zeroes into the seconds displays, and then using the B, C, D and E registers to set the clock. Study the flow diagram in Fig. 6-15 and the program listing below while we talk our way through it.

DIGITAL CLOCK

0100	21	LXI H,0008H	;Point H/L to display registers.
0101	08		;
0102	00		;
0103	36	MVI M,00H	;Send 00H to the display register No. 0.
0104	00		;
0105	23	INX H	;Increment H/L to point to register No. 1.
0106	36	MVI M,00H	;Send 00H to display register No. 1. We
0107	00		;have now loaded zeroes into the seconds.
0108	23	INX H	;Point H/L to space between displays.
0109	36	MVI M,11H	;Send blank between seconds and minutes.
010A	11		;
010B	23	INX H	;Point H/L to register No. 3.
010C	73	MOV M,E	;Load contents of E into register No. 3.
010D	23	INX H	;Point H/L to register No. 4.
010E	72	MOV M,D	;Load contents of D into register No. 4.
010F	23	INX H	;Point H/L to register No. 5.
0110	36	MVI M,11H	;Load blank between hours and minutes.
0111	11		;
0112	23	INX H	;Point H/L to register No. 6.
0113	71	MOV M,C	;Load contents of C into register No. 6.
0114	23	INX H	;Point H/L to register No. 7.
0115	70	MOV M,B	;Load contents of B into register No. 7.
0116	CD	CALL ONE SECOND (0320H)	;Display time for one second.

0117	20		;
0118	03		;
0119	21	LXI H,0008H	;Point H/L to display register No. 0.
011A	08		;
011B	00		;
011C	34	INR M	;Increment contents of register No. 0.
011D	7E	MOV A,M	;Move new contents of register to accum.
011E	FE	CPI 0AH	;Does accum now contain 0AH?
011F	0A		;
0120	C2	JNZ 0116H	;If not, go back to 0116H for another
0121	16		second.
0122	01		;
0123	3E	MVI A,00H	;If so, load accum with 00H.
0124	00		;
0125	77	MOV M,A	;Send the 00H back to the display reg.
0126	23	INX H	;Point H/L to reg No. 1.
0127	34	INR M	;Increment count in reg No. 1.
0128	7E	MOV A,M	;Bring new count to accum.
0129	FE	CPI 06H	;Does accum now contain 06H?
012A	06		;
012B	C2	JNZ 0116H	;If not, go back to 0116H for another
012C	16		second.
012D	01		;
012E	3E	MVI A,00H	;If so, load accum with 00H.
012F	00		;
0130	77	MOV M,A	;Send the 00H back to the display reg.
0131	23	INX H	;Point H/L to reg No. 3, minutes display.
0132	23	INX H	;
0133	34	INR M	.Increment count in reg No. 3.
0134	7E	MOV A,M	;Bring new count to accum.
0135	FE	CPI 0AH	;Does accum now contain 0AH?
0136	0A		;

0137	C2	JNZ 0116H	;If not, go back to 0116H for another
0138	16		second.
0139	01		;
013A	3E	MVI A,00H	;If so, load accum with 00H.
013B	00		;
013C	77	MOV M,A	;Send the 00H back to the display reg.
013D	23	INX H	;Point H/L to reg No. 4.
013E	34	INR M	;Increment count in reg No. 4.
013F	7E	MOV A,M	;Bring new count to accum.
0140	FE	CPI 06H	;Does accum now contain 06H?
0141	06		;
0142	C2	JNZ 0116H	;If not, go back to 0116H for another
0143	16		second.
0144	01		;
0145	3E	MVI A,00H	;If so, load accum with 00H.
0146	00		;
0147	77	MOV M,A	;Send the 00H back to the display reg.
0148	23	INX H	;Point H/L to reg No. 6, hours display.
0149	23	INX H	;
014A	34	INR M	;Increment count in reg No. 6.
014B	7E	MOV A,M	;Bring new count to accum.
014C	FE	CPI 0AH	;Does accum now contain 0AH?
014D	0A		;
014E	C2	JNZ 0116H	;If not, go back to 0116H for another
014F	16		second.
0150	01		;
0151	3E	MVI A,00H	;If so, load accum with 00H.
0152	77	MOV M,A	;Send the 00H back to the display reg.
0153	23	INX H	;Point H/L to reg No. 7.
0154	34	INR M	;Increment count in reg No. 7.
0155	7E	MOV A,M	;Bring new count to accum.
0156	FE	CPI 02H	;Does accum now contain 02H?

0157	02		;
0158	C2	JNZ 0116H	;If not, go back to 0116H for another
0159	16		second.
015A	01		;
015B	3E	MVI A,00H	;If so, load accum with 00H.
015C	00		;
015D	77	MOV M,A	;Send the 00H back to the display reg.
015E	C3	JMP 0116H	;Return to 0116H to set up loop.
015F	16		;
0160	01		;

Our program assumes that B, C, D and E registers are used to set the time. The displays are arranged such that the two right most display digits are initialized with zeroes as part of the opening portion of the program. Since these two digits are the seconds display, this means that the process of setting a time into the clock only sets the hours and minutes; seconds are set to 00 by the program. A different register is used for each of the displayed digits. Hence, to set the clock to 12:34, the B register would be set to 01, the C register to 02, the D register to 03, and the E register to 04. The **DCR** key is used to accomplish the setting of the registers; when the **EXC** key is pressed, the contents of these registers are loaded into the display registers and the clock started.

To initialize the display registers, the H and L registers are pointed to display register No. 0 with an **LXI H,0008H** instruction. Since this is the least significant of the seconds displays, we will load 00H into that memory location. We could do this by loading the accumulator with the data via a **MVI A,00H** followed by a **MOV M,A**. This would require three memory locations, two for the **MVI** and one for the **MOV**. There is a simpler way to do this by using indirect addressing. Since the H/L registers are already pointing to the destination of the data, a new instruction, **MVI M,00H**, can be used. This causes the data accompanying the instruction, in this case 00H, to be loaded into the memory location addressed by the H and L registers. The instruction operates in much the same fashion as the **MVI A** instruction that we are already familiar with. The major

difference is that the MVI A instruction is an immediate, directly addressed instruction, while the MVI M is an immediate, indirectly addressed instruction. This reduces the memory requirements from three locations to two locations.

The same data must be loaded into display register No. 1 to complete setting the seconds display to 00. This requires that the H/L pointer be incremented to point to the next location. With the instructions that we have discussed so far, this could be done with an INR L. If there was a chance that the incrementing process might take the L register from FFH to 00H, the H register will have to be involved so that the change becomes 00 FFH to 01 00H. Although this is not the case here, since the display registers are well within the range of the addressing capability of the L register, it is good programming practice to write the program so that the locations do not matter. Then, if locations are changed for some reason, the program will operate with a minimum of changes. If writing this insurance into the program requires extra instructions, the flexibility it provides may not be worth the extra memory locations. On the other hand, if the flexibility can be added without increasing the number of memory locations for the program, prudence dictates that we do so.

There are a set of instructions used to increment register pairs in the CPU, and we will use one, INX H, to do so now. Thus if H contains 00H and L contains FEH, executing an INX H will cause the H and L registers to contain 00H and FFH, respectively. A second INX H will cause them to contain 01H and 00H. Since the INX H instruction requires only one memory location, we have removed the restriction that the display registers be within the addressing capability of a single register, and can therefore be located anywhere in the memory. INX H is certainly a useful instruction to add to our repertoire, and we will be able to make good use of it. If you think back to the program we have written so far, there have been many occasions when we have incremented a register and used the zero flag as a sign to increment the second register. By using the INX instruction there is no longer a need to test the first register to see if the second should also be in-

cremented. Both are done at once. As with most good things, there is one catch. The INX instruction does not set any of the flags. This makes the instruction somewhat awkward when using it as a counter, since the test for zero operation is no longer quite so straight forward. In the present situation, there is no need to test for zero, so the INX instruction is ideal. When the INX H has been executed, H and L are pointing to display register No. 1, the most significant of the seconds display.

We want to separate the hours, minutes and seconds on the displays so that the numbers will not run together and will be easy to read. Since there are eight displays, allocating two of them to be blanks allows us to separate the displays into three fields. One of these blanks should be in display No. 2 so that the seconds and minutes are separated. An FFH in any display buffer will cause the corresponding digit display to be blank. Unfortunately we cannot merely load the buffer with FFH because we are planning to use CONVERT to load the seven-segment equivalents of the contents of the display register into the display buffers. Thus the contents of display register No. 2 will be converted and loaded into display buffer No. 2. Somehow we must load a number into the display register in a way that will cause it to be converted to an FFH in the display buffer. To do that we scan the lookup table at 8282H. Sure enough, at location 8293H is an FFH. Since this location is 11H locations above the first entry at 8282H, loading 11H into display register No. 2 will cause CONVERT to load FFH into display buffer No. 2. Perhaps we are making too much of a coincidence of finding an FFH in the lookup table. That entry is in there for the express purpose of allowing you to blank certain of the display digits. Likewise there are also entries for certain letters, dashes and decimal points in the table.

Displays No. 3 and 4 are to contain the minutes display. Since the clock is initially set by loading the correct minutes into the D and E registers with the **DCR** key, part of our initial program must load the contents of these registers into the proper display registers. To simplify matters, we assume that the most significant digit of the minutes is contained in the D register and the

least significant minutes digit in the E register. We point the H/L pair to display register No. 3 with an INX H, and then load the register into the display register with a MOV M,E. Another INX H followed by MOV M,D loads display register No. 4.

Between the hours and the minutes displays, another blank should appear. This is in display No. 5, and we implement that with an INX H and a MVI M,11H. The data for the hours displays is contained in the B and C registers. We load this data into display registers No. 6 and 7 with an INX H and a MOV M,C followed by a second INX H and MOV M,B. The clock is now set.

The portion of the program that we have just discussed is executed only once. The remainder of the program is a loop that is executed once every second. That loop begins by CALLing the one second delay subroutine at location 0320H. Since this subroutine takes exactly one second to execute, when the return to the main program occurs, it will be time to update the displays. We do this by incrementing the contents of the least significant seconds displays. An LXI H,0008H points H/L to the display register No. 0 and an INR M increments the contents of the register. During the one second subroutine, the contents of the display registers appear continuously on the displays, and the next time through the loop, the seconds display will appear as 01.

Each time we pass through the loop, the seconds display will be incremented. There is a limit to this process, however, since incrementing a seconds display of 09 will produce 0A. Since the world is not quite ready for hexadecimal clocks, we will obviously have to do something as the incrementing process goes beyond 09. The first step is to determine when that occurs. For this purpose we will use a class of instructions called COMPARE. All but one of these instructions compare the contents of one working register to the contents of the accumulator. If there is a match, the zero flag is set. If not match occurs, the zero flag is not set. It is up to the programmer to then check the status of the flag with either a conditional jump or call. We will use the second variety of compare instruction, CPI, Compare Immediate. In this instruction, the data

that accompanies the instruction is compared to the contents of the accumulator. As before, a match causes the zero flag to be set. We bring the contents of the display register to the accumulator with a `MOV A,M` and then check them with the `CPI` instruction. Since we are using the `CPI` instruction to monitor the incrementing process to detect the transition from 09 to 0A, the instruction should take the form `CPI 0AH`. That means that for the first nine incrementing operations, the zero flag will not be set since the accumulator will contain a number less than 0AH. However on the tenth pass through the loop, the display register will be incremented to 0AH, and when it is brought to the accumulator and checked with a `CPI 0AH`, a match will occur. The zero flag is set, and the next instruction, `JNZ 0116H`, will detect the change. The `JNZ` instruction has been causing program control to loop back to the one-second subroutine up until now. This has been causing the displays to appear for one second, change to the new seconds display for one second, then the next, and so on. When the `CPI 0AH` instruction sets the zero flag, however, the path changes. Now the `JNZ` test fails and program control falls through without executing the jump. We have successfully detected the transition from 09H to 0AH.

When we count in the decimal system, we do so by saying zero, one, two, three, four, five, six, seven, eight, nine. This corresponds to the written numerals 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. When we reach nine, a strange thing happens. We have run out of numerals, and, unless we are going to invent some more symbols, we have to come up with a way to count higher. The way, of course, is to introduce the concept of positional notation. That means that you begin using two numerals, where the left-most digit is understood to be a multiplier of tens. 10 thus becomes one ten and zero ones. 11 means one ten and one one. Later on, 34 means three tens and four ones.

We are faced with exactly the same thing in our digital clock program. When the `CPI` and `JNZ` combination failed, it told us that we had just run out of numerals and it was time to increment the next digit. First, however, we must load 00H back into display digit No. 0. After all, incrementing 09 produces 10, not 19! This is done with a `MVI A,00H` followed by a `MOV M,A`. Now

that the 00H is loaded into display No. 0, we go on to increment display No. 1. This is done with an INX H and an INR M. When the loop is completed, the instructions that we have just discussed will cause the seconds display to begin at 00 and tick off the seconds until 59 is reached. Actually the instructions we have just discussed will cause the displays to go right on ticking to 60, 70, 80, 90, A0, B0, etc. This is our way of pointing out that we have reached another of those points where we need to step in and interrupt the process.

By now, you should know the process. A MOV A,M brings the contents of the register to the accumulator where it can be tested with a CPI 06H. When the incrementing process causes the seconds displays to reach 60, the CPI 60H and JNZ 0116H that follows it fails. Program control falls through to another set of instructions that load 00H into the display register No. 1, and the contents of display register No. 3, which contains the least significant digit of the minutes display, is incremented. The rest of the program follows in the same vein, the only differences lie in the value contained in the CPI instruction used to determine when the incrementing process was getting out of bounds.

Go ahead and load the program. Remember to set in the correct time using the DCR key, then press CLR and EXC and watch the results. This should keep you fascinated for a while, so we'll wait for your attention to return before going on to something that's better yet. Since that something is an even better digital clock, we should warn you that we didn't spend a lot of time tweaking the timing constants in this first version of the clock. That's because we knew that super clock was coming on the next page. So don't worry if the accuracy of version one seems a little off after a while. Version two will be right on the button! See you when you're done playing.

